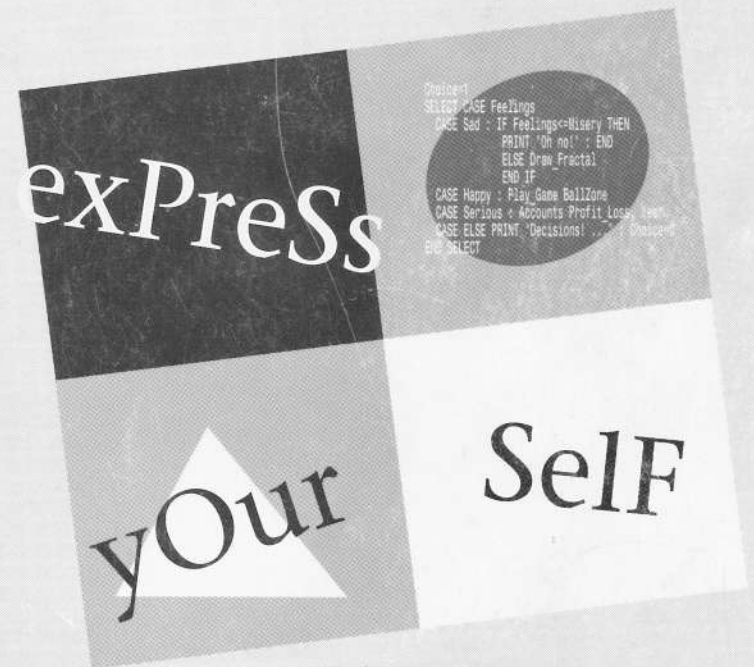


# HiSoft BASIC

version 2



**Technical  
Reference**

HiSoft BASIC 2 for  
ST/STE/TT Computers

ISBN 0 948517 49 2

**HiSoft**  
High Quality Software

more software for your Atari  
ST/STE/TT from

**HiSoft**  
High Quality Software

# **Technical Reference**

## **HiSoft BASIC 2 for the Atari ST/STE/TT**

**By HiSoft**

© Copyright 1991 HiSoft. All rights reserved.

**Program:** designed and programmed by HiSoft.

**Manual:** written by David Nutkins, Alex Kiernan and Tony Kendle.

This guide and the HiSoft BASIC 2 program diskettes contain proprietary information which is protected by copyright. No part of the software or the documentation may be reproduced, transcribed, stored in a retrieval system, translated into any language or transmitted in any form without express prior written consent of the publisher and copyright holder(s).

HiSoft shall not be liable for errors contained in the software or the documentation or for incidental or consequential damages in connection with the furnishing, performance or use of the software or the documentation.

HiSoft reserves the right to revise the software and/or the documentation from time to time and to make changes in the content thereof without the obligation to notify any person of such changes.



**HiSoft**  
High Quality Software

*Published by HiSoft*

*The Old School, Greenfield, Bedford MK45 5DE UK*

First Edition, November 1991 - ISBN 0 948517 49 2

# Table of Contents

---

|  |            |
|--|------------|
| <b>Chapter 1 Command Reference</b>       | <b>1</b>   |
| <b>Chapter 2 Library Reference</b>       | <b>209</b> |
| <b>Introduction</b>                      | <b>209</b> |
| Important Note                           | 210        |
| <b>The Libraries</b>                     | <b>210</b> |
| The GEMDOS Library                       | 210        |
| The BIOS library                         | 217        |
| The XBIOS Library                        | 220        |
| STESOUND library                         | 229        |
| <b>The GEMVDI library</b>                | <b>230</b> |
| GEMVDI : Control                         | 232        |
| GEMVDI : Drawing Primitives              | 234        |
| GEMVDI: Attributes                       | 237        |
| GEMVDI: Raster Functions                 | 242        |
| GEMVDI: Mouse & Keyboard State Functions | 243        |
| GEMVDI: Enquires                         | 244        |
| <b>The GEMAES Library</b>                | <b>247</b> |
| GEMAES: Utility routine                  | 248        |
| GEMAES: Message Passing                  | 248        |
| GEMAES: Events                           | 248        |
| GEMAES: Menus                            | 251        |
| GEMAES: Objects                          | 252        |
| GEMAES: Forms                            | 254        |

|                                 |            |
|---------------------------------|------------|
| GEMAES: Graphics                | 255        |
| GEMAES: Scrap Directory         | 257        |
| GEMAES: File Selector           | 258        |
| GEMAES: Windows                 | 258        |
| GEMAES: Resource Files          | 263        |
| GEMAES: Shell Routines          | 264        |
| <b>Menu Library</b>             | <b>265</b> |
| <b>Chapter 3 HGT Reference</b>  | <b>267</b> |
| <b>General control routines</b> | <b>267</b> |
| <b>Object Handling routines</b> | <b>269</b> |
| Menu routines                   | 273        |
| Dialogs and Alerts              | 274        |
| Other object routines           | 275        |
| <b>The Window system</b>        | <b>276</b> |
| Object Windows                  | 278        |
| Image window routine            | 278        |
| Text windows                    | 278        |
| <b>User Desktop routines</b>    | <b>280</b> |
| <b>File Selector Routines</b>   | <b>281</b> |
| <b>Running other programs</b>   | <b>281</b> |
| <b>Image routines</b>           | <b>282</b> |
| <b>Desk Accessories</b>         | <b>283</b> |

|                                       |            |
|---------------------------------------|------------|
| <b>Chapter 4 WERCS Reference</b>      | <b>285</b> |
| <b>Objects</b>                        | <b>285</b> |
| Flag Types                            | 289        |
| Flag States                           | 291        |
| Object, Flags and States Summary      | 292        |
| <b>Programming with Resources</b>     | <b>294</b> |
| Tree Structure                        | 294        |
| Hints & Tips on Resources             | 303        |
| Common Mistakes and how to avoid them | 304        |
| .BH file details                      | 306        |
| <b>The WTEST Program</b>              | <b>306</b> |
| WTEST structure                       | 306        |
| <b>Chapter 5 MonSTB Reference</b>     | <b>309</b> |
| <b>Numeric Expressions</b>            | <b>309</b> |
| Numbers                               | 310        |
| Character Constants                   | 310        |
| Symbols and Registers                 | 311        |
| <b>Window Types</b>                   | <b>311</b> |
| Register Window Display               | 312        |
| Disassembly Window Display            | 313        |
| Memory Window Display                 | 313        |
| Source-code Window Display            | 313        |
| <b>Window Commands</b>                | <b>314</b> |
| <b>Screen Switching</b>               | <b>317</b> |
| <b>Breaking into Programs</b>         | <b>318</b> |

|  |            |
|--|------------|
| <b>Breakpoints</b>                             | <b>318</b> |
| <b>History</b>                                 | <b>321</b> |
| <b>Quitting MonSTB</b>                         | <b>321</b> |
| <b>Loading &amp; Saving</b>                    | <b>322</b> |
| <b>Executing Programs</b>                      | <b>323</b> |
| <b>Searching Memory</b>                        | <b>325</b> |
| <b>Miscellaneous</b>                           | <b>327</b> |
| <b>Command Summary</b>                         | <b>331</b> |
| Window Commands                                | 331        |
| Screen Switching                               | 331        |
| Breakpoints                                    | 331        |
| Loading and Saving                             | 331        |
| Executing Programs                             | 332        |
| Searching Memory                               | 332        |
| Miscellaneous                                  | 332        |
| <b>Debugging Stratagem</b>                     | <b>332</b> |
| Hints & Tips                                   | 332        |
| Bug Hunting                                    | 333        |
| <b>Exceptions</b>                              | <b>334</b> |
| <b>Memory Layout</b>                           | <b>336</b> |
| Exception Analysis                             | 337        |
| <b>Using MonSTB with other languages</b>       | <b>338</b> |
| <b>Using MonSTB with multi-module programs</b> | <b>338</b> |
| <b>For Devpac MonST2 Users</b>                 | <b>338</b> |

|                                     |            |
|-------------------------------------|------------|
| <b>Appendix A Reserved Words</b>    | <b>341</b> |
| <b>Appendix B Compiler Options</b>  | <b>343</b> |
| <b>Appendix C Error Messages</b>    | <b>345</b> |
| <b>GEMDOS Error Numbers</b>         | <b>345</b> |
| <b>Runtime Errors</b>               | <b>346</b> |
| Run-time Errors Alphabetically      | 347        |
| <b>Compilation Errors</b>           | <b>350</b> |
| Compiler Error Messages             | 351        |
| <b>Appendix D The ASCII Table</b>   | <b>363</b> |
| <b>Appendix E WERCS .HRD format</b> | <b>365</b> |
| HRD Header record                   | 365        |
| HRD Data Record                     | 366        |
| <b>Appendix F Linking</b>           | <b>367</b> |
| <b>Code Generation</b>              | <b>367</b> |
| Register Usage                      | 367        |
| <b>Startup Options</b>              | <b>370</b> |
| <b>Low Level Debugging</b>          | <b>370</b> |
| Finding Your Way                    | 370        |
| <b>The Heap and Descriptors</b>     | <b>371</b> |

|                                      |            |
|--------------------------------------|------------|
| <b>Memory Formats</b>                | <b>372</b> |
| Single-precision Floating Point      | 372        |
| Double-precision Floating Point      | 372        |
| <b>Linking or Libraries?</b>         | <b>373</b> |
| Assembler utility routines           | 373        |
| Linking                              | 374        |
| External Definition                  | 374        |
| Linking                              | 376        |
| How to write a Library               | 378        |
| The Actual Code                      | 381        |
| Strings and Things                   | 383        |
| Returning values from Functions      | 383        |
| Library Format Summary               | 384        |
| <b>Using BUILDLIB</b>                | <b>384</b> |
| STESOUND library                     | 385        |
| <b>Appendix G Desk Accessories</b>   | <b>389</b> |
| <b>Get Ready</b>                     | <b>389</b> |
| <b>Commands to use</b>               | <b>390</b> |
| FUNCTION menu_register%              | 390        |
| WINDOW SYSTEM                        | 390        |
| WINDOW STOP                          | 391        |
| <b>Structure of a Desk Accessory</b> | <b>391</b> |
| <b>Restrictions</b>                  | <b>391</b> |

|                                     |            |
|-------------------------------------|------------|
| <b>Appendix H Program Profiling</b> | <b>393</b> |
| <b>Introduction</b>                 | <b>393</b> |
| <b>Creating the Profile</b>         | <b>393</b> |
| <b>Analysing the Profile</b>        | <b>394</b> |
| <b>Appendix I TT Specifics</b>      | <b>395</b> |
| <b>TT version of the compiler</b>   | <b>395</b> |
| <b>TT Libraries</b>                 | <b>396</b> |
| <b>TT Debugger</b>                  | <b>396</b> |



# Chapter 1

## Command Reference

---

This chapter gives a detailed description of each and every HiSoft BASIC statement and function.

It is arranged as follows:

- **Syntax**

This shows the allowable forms of the statement or function.

Parameters are denoted in *italics* and optional items are enclosed in square brackets [ ].

- **Effect**

Details a summary of the actions of the statement or function.

- **Comments**

This describes the actions in much greater detail, where this is useful.

- **See**

Any entries under this heading will point you towards other commands that are related in meaning to this command. It will normally be useful to consult these other entries for background information.

- **Example**

Shows one or more examples of the statement or function in use.

## ABS function

---

- **Syntax**

ABS(*numeric\_expression*)

- **Effect**

This function returns the absolute value of the *numeric\_expression*

- **Comments**

The absolute value function returns the unsigned value of the *numeric\_expression*. The absolute values of both -1 and 1 is 1. The type of the result is the same as the type of *numeric\_expression*.

- **See**

SGN.

- **Example**

```
PRINT ABS(6*(7)), ABS(6*(-7))
```

Result:

42 42

## ASC function

---

- **Syntax**

ASC(*string\_expression*)

- **Effect**

This function returns a numeric value that is the ASCII code for the first character of the *string\_expression*.

- **Comments**

If the string passed is a null-string, an Illegal function call error is given. The result is an integer.

- **See**

CHR\$.

- **Example**

```
X$="FORTY-TWO"  
PRINT ASC(X$)
```

'Prints ASCII code for "F"

Result:

70

## ATN function

---

- **Syntax**

ATN(*numeric\_expression*)

- **Effect**

This function returns the arctangent of the *numeric\_expression*, or the angle whose tangent is the *numeric\_expression*.

- **Comments**

The result is returned in radians, in the range of  $-\pi/2$  to  $\pi/2$  radians.

The *numeric\_expression* can be of any numeric type. ATN is single precision by default; if the numeric value is double precision, ATN returns a double precision value.

- **See**

SIN, TAN, COS.

- **Example**

```
PRINT 4*ATN(1);4*ATN(1#)
```

Result:

3.141593    3.141592653589794

## BAR statement

---

- **Syntax**

BAR *x\_start*,*y\_start*,*width*,*height*

- **Effect**

An outlined box is drawn in the current output window.

- **Comments**

The same drawing attributes as CIRCLE are used, i.e. the current fill colour is used for the outline, and together with the current index and style it is used for the interior.

*x\_start* and *y\_start* specify the top-left corner of the bar relative to the current output window.

This call uses the GEMVDI *v\_bar* graphics primitive.

- **See**

COLOR, LINEF, *v\_bar* VDI command.

- **Example**

```
COLOR 1,1,1,2  
BAR 100,50,50,20
```

## BEEP statement

---

- **Syntax**

BEEP

- **Effect**

The monitor speaker will beep.

- **Comments**

BEEP uses the ASCII bell character; the same effect can be attained by using PRINT CHR\$(7);.

- **See**

SOUND, WAVE.

- **Example**

```
FOR I=1 TO 10
    BEEP
NEXT I
```

This loop produces a single beep ten times the length of a one BEEP.

## BIN\$ function

---

- **Syntax**

BIN\$(*numeric\_expression*)

- **Effect**

This function returns a string that represents the binary value of the *numeric\_expression*. If the expression is an integer then the resulting string will be from 1 to 16 characters in length, but if it the expression is a long integer the result can be up to 32 characters long.

- **Comments**

The *numeric\_expression* is rounded before being evaluated by HEX\$. If the parameter is an integer then the resulting string will be from 1 to 4 characters in length, and if it is a long integer then the result may be up to 8 characters in length.

- **See**

OCT\$, HEX\$, *User Manual : Concepts* chapter.

- **Example**

```
PRINT BIN$(8);
```

Result:

1000



## BLOAD statement

### • Syntax

BLOAD *filename*,*address*

or

BLOAD #*channel*,*address*,*byte\_count*

### • Effect

This statement loads a binary file into the buffer specified, or reads an arbitrary number of bytes from the given channel.

### • Comments

The *filename* is a string expression following the standard GEMDOS conventions. The *address* is a long integer.

The buffer address is the responsibility of the user. BLOAD does not check if the address given is a safe address in RAM.

In the second form the channel should be opened using the OPEN statement and *byte\_count* gives the number of bytes to read. This form of BLOAD is similar to the INPUT\$ except that it lets you read into an array, or another address, instead of a string.

### • See

BSAVE, INPUT\$, GET.

### • Example

```
BLOAD "PICTURE.DMP",SPEEKL(&H44E)
```

This line loads a binary file directly into the video-RAM (which for the standard ST resolutions is 32000 bytes in length). &H44E is the system variable where the address of the screen is kept. SPEEKL is necessary, as the address of the video-RAM is 32 bits long and the system variable is in supervisor memory.

```
OPEN "PICTURE.NEO" FOR INPUT AS #1
DIM header(64)
BLOAD #1,VARPTR(header(0)),128
BLOAD #1,SPEEKL(&H44E),32000
CLOSE #1
```

This example will load a NEOchrome file directly into the video RAM for ST low resolution, reading the Neochrome file header into the array `header()`. The LOADSCR.BAS program is another example of its use.

## BSAVE statement

### • Syntax

BSAVE *filename*,*address*,*length*

or

BSAVE #*channel*,*address*,*length*

### • Effect

This statement saves the contents of the specified buffer to the output device specified in *filename* or #*channel*.

### • Comments

It is possible, though not necessarily useful, to BSAVE to the parallel or serial ports. Both the address as well as the length of the buffer are long integers. The *filename* must follow the GEMDOS conventions; alternatively you can output to a file that is already opened.

### • Example

```
BSAVE "PICTURE.DMP",PEEK(&H44E&),32000
```

This line saves the entire ST video RAM to a disk file named PICTURE.DMP.

```
DIM a%(100),b%(100)
```

```
...
```

```
OPEN "ARRAYS.DAT" FOR OUTPUT AS #1
```

```
BSAVE #1,VARPTR(a%(0)),101*2 '101 elements of 2 bytes each
```

```
BSAVE #1,VARPTR(b%(0)),101*2
```

```
CLOSE #1
```

This program fragment saves the entire contents of the arrays `a%()` and `b%()` to the file ARRAYS.DAT. Note that you can only save numeric arrays in this manner.

### • See

BLOAD, PRINT#, WRITE.

## CALL statement

### • Syntax

[CALL] sub\_program\_name [(parameter [,parameter]...)]

### • Effect

Calls a sub-program defined using SUB...END SUB or a library routine.

### • Comments

sub\_program\_name is the name of the sub-program being called. There must be the same number of parameters as in the sub-program's declaration.

Further information on sub-programs can be found in the *Concepts* chapter of the *User Manual*.

The keyword CALL can always be omitted if the sub-program has parameters and if it is not the first statement of a multi-statement line; this is to avoid conflicts with label declarations which are also identifiers followed by a colon. If CALL is omitted the brackets round the parameters must be omitted.

Array parameters must be passed array names followed by ( ) in CALL statements and string parameters must be passed strings. However different numeric types will be converted to the type required by the sub-program.

Parameters are passed by reference if they consist of just a variable of the same type as in the declaration of the sub-program and the VAL or BYVAL keywords were not specified for the parameter when the sub-program was defined. Array elements may be passed as variable parameters.

You can pass a variable by value when it would otherwise be passed by reference simply by enclosing the parameter in parentheses.

### • See

SUB, FUNCTION.

### • Examples

```
QuickSort MyArray(),100
CALL QuickSort(MyArray(),100)      ' these are equivalent
QuickSort(MyArray(),100)           ' syntax error because
                                   ' of brackets

call SKIP_ONE:SKIP_ONE 'call required for first call
FRED X                        ' call by reference
FRED (X)                      ' call by value
```

## CALL LOC statement

### • Syntax

CALL LOC address [,parameter]...

### • Effect

Call a machine-code routine, with the option of passing parameters.

### • Comments

This is directly equivalent to the ST BASIC CALL statement. Unfortunately we had to change the syntax to avoid confusion between machine-code and sub-program CALL statements.

The machine-code at address is executed, and these parameters can be found on the stack:

|       |      |                            |
|-------|------|----------------------------|
| (a7)  | long | return address             |
| 4(a7) | word | number of parameters       |
| 6(a7) | long | pointer to parameter block |

The parameter block consists of a sequence of longs, one for each parameter passed. In the case of numeric parameters this is the rounded long-integer value of the parameter, but in the case of string parameters this is a pointer to the string, null terminated.

The machine-code can return to the BASIC program by executing a RTS statement. All registers except A7 may be destroyed.

In general it is better to access machine code by linking with the machine code routine or writing your own HiSoft BASIC library.

### • See

Appendix F: Linking, Writing your own libraries.

### • Example

```
DIM spare%(100)
BLOAD "CALCPI.PRG",VARPTR(spare%(0))
calcpi&=VARPTR(spare%(0))+28      '+28 to skip header
CALL LOC calcpi&,100,"PI"
```

## **CALLS statement**

### • **Syntax**

CALL *sub\_program\_variable*

### • **Effect**

Calls a sub-program indirectly using a variable as a pointer for it.

### • **Comments**

*sub\_program\_variable* must have been initialised using the VARPTRS function to point to a sub-program. the sub-program must not have any parameters.

Caution: this is for advanced programmers only. Incorrect use of CALLS can wreak havoc on a running program or its memory area.

### • **See**

VARPTRS, CALL.

### • **Example**

```
read_char&=VARPTRS(char_from_disc)
DO
    CALLS read_char&
LOOP UNTIL no_more
.
.
SUB char_from_disc
IF EOF(2) THEN
    read_char&=VARPTRS(char_from_mem)
    CALLS read_char&
EXIT SUB
ELSE
.
.
.
END SUB
```

## **CDBL function**

### • **Syntax**

CDBL(*numeric\_expression*)

### • **Effect**

This function converts the *numeric\_expression* to a double precision number.

### • **Comments**

The effect is identical to assigning the *numeric\_expression* to a double precision variable and then using the double precision variable.

### • **See**

MKD\$, CINT, CLNG, CSNG.

### • **Example**

```
PRINT ATN(1)           'single precision
PRINT CDBL(ATN(1))     'single extended to double
PRINT ATN(CDBL(1))     'double precision accuracy
```

Result:

```
.7853982
.7853981852531433
.7853981633974485
```

## CHAIN statement

- **Syntax**

CHAIN *filename*

- **Effect**

This statement loads and executes another program.

- **Comments**

*filename* should conform to GEMDOS specifications and be the name of an executable (i.e. double-clickable) program, including the extension. The program does not have to be a HiSoft BASIC compiled program.

The type of the program being CHAINED should be the same as the type of the program doing the CHAIN. For example trying to CHAIN a .PRG file from within a .TOS program will produce unpredictable and potentially disastrous results. CHAIN should not be used when compiling to memory.

All files are closed before the program is CHAINED.

- **See**

RUN, Execute (HiSoft GEM Toolbox routine).

- **Example**

```
DO
  INPUT "File to run";f$
  f$=f$+".PRG"
  LOOP UNTIL FEXISTS(f$)
  CHAIN f$
```

## CHDIR statement

- **Syntax**

CHDIR *pathname*

- **Effect**

This statement changes the current directory.

- **Comments**

The *pathname* specified must conform to the GEMDOS conventions. This statement does not change the default drive, even if a drive is specified.

It is possible to use relative (as opposed to absolute) path names. If a directory \ONE\TWO exists and it is the current directory, CHDIR ".." will change the current directory to one hierarchic level higher (in this case \ONE).

It is also possible to CHDIR one level downwards. To switch from \ONE to \ONE\TWO requires CHDIR "TWO"; CHDIR "\ONE\TWO" is not necessary in this case.

- **See**

CURDIR\$.

- **Example**

```
CHDIR "\HBASIC" 'an absolute path
CHDIR "SOURCE"  'this changes the directory
                  'to \HBASIC\SOURCE
```



## CHR\$ function

- **Syntax**

CHR\$(*ASCII\_code*)

- **Effect**

This function returns a one character string whose *ASCII\_code* was passed as the parameter.

- **Comments**

CHR\$ is usually used to produce characters which are not readily available from the keyboard. Common uses are sending a form feed (ASCII 12) to a printer or printing foreign characters on the screen. The *ASCII\_code* is an integer.

- **See**

ASC, User Manual : Concepts: Character Constants.

- **Example**

```
LPRINT CHR$(12);      'sends a form feed to the printer
PRINT "Copyright ";CHR$(189);" Acme Programming"
```

## CINT function

- **Syntax**

CINT(*numeric\_expression*)

- **Effect**

This function converts the *numeric\_expression* to an integer value by rounding its fractional part.

- **Comments**

If the *numeric\_expression* is not in the range -32768 to 32767, an Overflow error is returned.

CINT differs from INT and FIX in that it produces an integer value by rounding. An example of the differences of the three functions can be found under INT.

This is equivalent to assigning the *numeric\_expression* to an integer variable and then using that variable.

- **See**

INT, FIX, MKIS\$, CDBL, CSNG, CLNG.

- **Example**

```
PRINT CINT(1.5),
PRINT CINT(-1.5)
```

Result:

2 -2

## CIRCLE statement

### • Syntax

CIRCLE *x\_centre,y\_centre,radius* [,*start\_angle,end\_angle*]

### • Effect

Draws a circle or arc in the current window.

### • Comments

If the optional angle parameters are omitted, the command draws a hollow circle in the current line colour (the third parameter in the COLOR statement). The angle parameters are expressed in degrees times 10, and are measured anti-clockwise starting on the right side. Solid circles and arcs can be drawn using the PCIRCLE command. All the parameters are integers.

The *x\_centre* and *y\_centre* are relative to the current window rather than the screen. CIRCLE ensures that the x radius and y radius are the same in pixels. As a result the graphic that is drawn will not appear circular in some resolutions (e.g. ST Medium) where the pixels are not square. To draw circles that look like circles but which will have different y-radii use the GEMVDI *v\_circle* command.

### • See

ELLIPSE, PCIRCLE, BAR, *v\_circle*, *v\_ellarc*, *v\_ellipse* (VDI routines).

### • Example

```
CIRCLE 200,100,75
CIRCLE 100,100,50,450,325
```

## CLEAR statement

### • Syntax

CLEAR

### • Effect

This statement clears all variables and closes all channels.

### • Comments

The following actions are taken by this command:

- All global numeric variables are reset to zero
- All global string variables are reset to a null string (i.e. " ")
- All numeric arrays have their contents reset to zero
- All string arrays have their contents reset to a null string
- All files are closed

This is provided for compatibility with old dialects of BASIC. In general we recommend that you don't use it - just clear what you need to instead.

### • Example

```
IF a$="ZERO" THEN CLEAR
```

## CLOSE statement

---

- **Syntax**

CLOSE [#]channel\_number[, [#]channel\_number]...

- **Effect**

This statement terminates I/O to the specified file or device.

- **Comments**

CLOSE is the opposite of OPEN. The *channel\_number* is the number specified in the OPEN statement for a particular file or device.

CLOSE without parameters closes all OPENed files and devices.

Once a file is CLOSEd, its channel number may be used to OPEN any unopened other file or device.

Closing a file or device that was opened for sequential output causes the final buffer to be written before closing.

If you have set SYSTAB+67 to a non-zero value when closing a sequential output file then a Ctrl-Z character (CHR\$(26)) will be appended to it

CLOSEing a file that was opened for random access with automatically clear all the FIELDed variables that were associated with this file.

CLEAR, END/SYSTEM, RESET, STOP and RUN close all files and devices automatically.

- **See**

OPEN, SYSTAB.

- **Example**

CLOSE #1, #2

## CLS statement

---

- **Syntax**

CLS

- **Effect**

Clears the screen and returns the cursor to the upper left corner.

- **Comments**

In a GEM program this will clear the current output window.

## COLOR statement

### • Syntax

COLOR *text\_colour* [, *fill\_colour*] [, *line\_colour*] [, *index*] [, *style*]

### • Effect

Sets the current colour and line drawing attributes.

### • Comments

This sets the colour of the text subsequently output, as well as the the fill colour, the colour of lines drawn and the fill index and style. All parameters except *text\_colour* are optional. In TOS programs only the first two parameters are acted upon. The following tables show the result of different parameters without the control panel present:

| Number | Colour     | Number | Colour       |
|--------|------------|--------|--------------|
| 0      | White      | 8      | Grey         |
| 1      | Black      | 9      | Dark grey    |
| 2      | Red        | 10     | Light blue   |
| 3      | Green      | 11     | Blue green   |
| 4      | Blue       | 12     | Light purple |
| 5      | Dark blue  | 13     | Dark purple  |
| 6      | Brown      | 14     | Dark yellow  |
| 7      | Dark green | 15     | Light yellow |

TT Low resolution                      colours 0-255

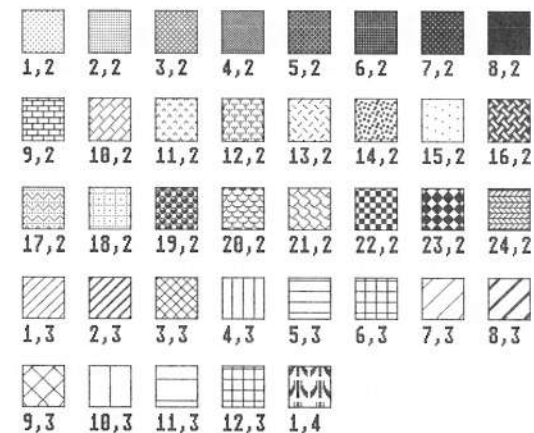
TT Medium & ST Low resolution:    colours 0-15

ST Medium resolution:                colours 0-3

ST & TT High resolution:            colours 0-1

The actual colours displayed may differ from those above, depending on the current palette setting. To ensure the colours appear as above use the PALETTE statement.

If the *style* parameter is 0 then all fill operations will be in the text colour, or if it is 1 then they will be in the background colour. Values of *style* from 2 to 4 inclusive act together with *index* to produce the different fill patterns shown on the next page:



Index and Style Combinations

A *style* of 4 gives the same pattern regardless of *index*.

### • Example

COLOR 1,1,1,1,4 ' Atari pattern  
PCIRCLE 200,100,80

## COMMAND\$ function

---

- **Syntax**

COMMAND\$

- **Effect**

This function returns the command line from the basepage of the program.

- **Comments**

This function enables a program to access the command line entered in the window produced by a .TTP or a GEM application installed as *TOS takes parameters* or with a particular file type. For example if you install MINE.PRG to accept .TXT files and double click on a .TXT file, the file's name is passed to MINE.PRG via COMMAND\$.

Any leading spaces are removed from the command line and it is upper-cased.

- **See**

ENVIRON\$.

- **Example**

```
OPEN COMMAND$ AS #1      ' a filename passed on the command
                          ' line is used to open a file
```

## COMMON SHARED statement

---

- **Syntax**

COMMON SHARED *variable* [ (*subscripts...*) ][ , *variable* [ (*subscripts...*) ] ]...

- **Effect**

This statement indicates that global variables may be accessed throughout any sub-programs.

- **Comments**

This statement is provided mainly for QuickBASIC compatibility.

It has the same effect as the DIM SHARED statement except that the subscripts of arrays are ignored. Arrays used in this statement should be DIMmed before the COMMON SHARED statement.

- **Example**

```
DIM glob(20)
COMMON SHARED debug_flag,glob(1)
```

is equivalent to

```
DIM SHARED debug_flag,glob(20)
```

```
' both debug_flag and glob() can now be accessed anywhere
' in the 'program.
```

## CONST statement

### • Syntax

CONST *name*=*integer\_constant* [,*name*=*integer\_constant*]...

### • Effect

CONST defines symbolic constant values for use in place of integer variables.

### • Comments

Constants must be declared before they are used and their names must be valid integer variable names. *integer\_constant* must be a simple integer number or character constant optionally preceded by a minus sign.

Constants can be used to make code that accesses data structures in memory much easier to read and also to give a loose equivalent to user enumerated types in Pascal.

Constants may be used within sub-programs and user-defined functions without declaring them as SHARED. Note that this is *not* the same as the latest MicroSoft BASIC *s* where constants are local to sub-programs.

### • See

DEFINT, *User Manual* : *Concepts* chapter.

### • Example

```
DEFINT a-z
CONST white=0,black=1,red=2,green=3
COLOR red,green,black
CONST firstletter="A",lastletter="Z"
IF (c%>=firstletter) AND (c%<=lastletter) THEN
    PRINT "It's a lower case letter"
```

## COS function

### • Syntax

COS(*numeric\_expression*)

### • Effect

This function returns the cosine of the *numeric\_expression*, which must be in radians.

### • Comments

COS is normally performed in single precision. If the *numeric\_expression* is double precision, COS is performed in double precision.

### • See

SIN, TAN, ATN.

### • Example

```
PRINT COS(0)
```

Result:

1

## CSNG function

---

- **Syntax**

CSNG(*numeric\_expression*)

- **Effect**

This function converts *numeric\_expression* to a single precision number.

- **Comments**

CSNG has the same effect as assigning *numeric\_expression* to a single precision variable and then using that variable.

- **See**

MK\$\$, CDBL, CINT, CLNG.

- **Example**

```
PRINT SIN(1#)
PRINT CSNG(SIN(1#))
```

Result:

```
.8414709848078965
.841471
```

## CSRLIN function

---

- **Syntax**

CSRLIN

- **Effect**

This function returns the current line position of the cursor.

- **Comments**

To return the current column position, use POS. The value returned is an integer, with the top of the screen (or window) returning the value of 1, corresponding to the LOCATE statement.

- **See**

POS, PRINT, LOCATE.

- **Example**

```
L%=CSRLIN
C%=POS(0)
PRINT "The cursor was on line";L%;"and in column";C%
```

Result (if the cursor is on line 2, column 3):

The cursor was on line 2 and in column 3



## CURDIR\$ function

### • Syntax

CURDIR\$ [(string\_expression)]

### • Effect

This function returns the current directory for a given drive or the current drive and directory if no *string\_expression* is specified.

### • Comments

The string returned always includes the drive name which ever form is used and is never terminated by a backslash (\).

Only the first character of the *string\_expression* is considered and this is not case sensitive; thus CURDIR\$("a") will return the same value as CURDIR\$("A").

If the string passed is null then the current directory and drive are returned.

### • See

CHDIR.

### • Example

```
' assume the current drive is C and HBASIC is the current
' directory on that drive and that the current directory
on
' drive B is the root directory
PRINT CURDIR$,CURDIR$("B:"),CURDIR$("c")
```

Result:

C:\HBASIC      B:      C:\HBASIC

## CVD,CVI,CVL,CVSfunctions

### • Syntax

CVD(8-byte string of a double precision float)

CVI(2-byte string of an integer)

CVL(4-byte string of a long integer)

CVS(4-byte string of a single precision float)

### • Effect

These functions return the internal numeric values of strings of bytes.

### • Comments

These functions are the counterparts to MKD\$,MKI\$,MKL\$ and MKS\$. They do not change the value of the actual data, they only induce BASIC to interpret them differently.

The most common use for these functions is for processing random access files in which numeric values have been stores as strings by the MKD\$,MKI\$,MKL\$ or MKS\$ functions.

### • Example

```
D#=9/11
I%=10
L&=42
S!=5/7
PRINT D#,I%,L&,S!
OPEN "TEST.DAT" FOR OUTPUT AS #1
PRINT #1,MKD$(D#);MKI%(I%);MKL(L&);MKS(S!)
CLOSE #1
'The converted values of the different variables are
'binary. Printing these values to screen would result in
'strange looking control characters being displayed.
OPEN "TEST.DAT" FOR INPUT AS #1
D1$=INPUT$(8,#1)
I1$=INPUT$(2,#1)
L1$=INPUT$(4,#1)
S1$=INPUT$(4,#1)
CLOSE #1
PRINT CVD(D1$),CVI(I1$),CVL(L1$),CVS(S1$)
```

Result:

```
.8181818127632141 10 42 7142857
.8181818127632141 10 42 7142857
```

## DATA statement

### • Syntax

DATA *constant* [,*next\_constant*]...

### • Effect

This statement defines the data to be used by the READ statement.

### • Comments

DATA is not executed. It may contain as many constants as will fit on a line. READ will read DATA in the order of their appearance in the program. Multiple DATA statements will be processed as one continuous string of constants.

Constants may be of any variable type. If necessary the data will be converted into the correct type for the variable being READ.

It is not necessary for quotation marks to be present around string constants in DATA unless the string contains colons, commas, leading and trailing spaces or if it might be misconstrued as a inappropriate numeric value.

HiSoft BASIC stores data items internally as the appropriate type, thus 1.12345 will be stored as a single precision number requiring only 4 bytes of storage. Since you will normally read this into a single precision variable there is no need to convert it from ASCII to binary at run-time unlike with many BASICs. Should you read a number into a string variable it will be converted back into a string.

This does have the disadvantage that the string representation may not be the same as the original. For example, 11E1, will come out as 110. You can avoid this problem by enclosing such strings in quotes.

A null item in the list of constants is allowed. When read, a numeric variable it is assigned the value 0. If read as a string variable, the value will have the null value.

READ can process DATA statements again if RESTORE is used. Otherwise it is not possible to re-read DATA.

### • See

READ, RESTORE, OPEN, INPUT\$.

### • Example

```
DATA 2.2,1.6,"Hello"  
READ a,b%,c$  
PRINT a;b%;c$
```

Result:

2.2 2 Hello

The data item 1.6 is rounded before being assigned to the integer b%, hence the result of 2.

## DATE\$statement/function

### • Syntax

DATE\$

### • Effect

This function returns the current date. If used as a statement, it can set the current date.

### • Comments

The returned value is a ten character string. The format is *mm-dd-yyyy*. When setting the current date the following formats may be used:

```
mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy
```

Note that unlike previous versions of HiSoft BASIC the year may be any date from 1970 to 2069.

### • See

TIME\$, tgetdate% & tsetdate% GEMDOS functions.

### • Example

```
d$=DATE$
IF RIGHT$(d$,4)>="1991" THEN
  PRINT "Today's date is: ";d$
ELSE
  PRINT "The system clock has not been set today"
  INPUT "Please enter today's date (mm-dd-yy): ";d$
  DATE$=d$
END IF
```

## DECLARE statement

### • Syntax

DECLARE [SUB|FUNCTION]routine\_name [(parameter\_list)]

or

DECLARE [SUB|FUNCTION]routine\_name [CDECL][ALIAS  
"externalname"][(parameter\_list)]

### • Effect

Declares a BASIC sub-program or function before it is defined or declares an externally linked sub-program or function..

### • Comments

When used without CDECL this declares a BASIC sub-program or function so that you can use it before it is defined. The routine\_name and parameter list should appear in exactly the same form as in the definition in the SUB or FUNCTION statement. The use of DECLARE for sub-programs is optional in HiSoft BASIC but is required in the latest MicroSoft BASICs. All dialects require FUNCTIONs to be defined before they are used to avoid confusion with ordinary variables or arrays.

If you DECLARE a function or sub-program but do not define it the compiler will give an error 48 unless you have used the Allow undefined sub-programs option ({}).

When CDECL is included this statement declares an external routine, written in C or assembler that will be linked with the program. The optional parameter list is of exactly the same format as for a BASIC sub-program or function. The use of this command causes HiSoft BASIC to generate a Lattice format linkable file rather than an executable one. See Appendix F for more details.

The name of the external routine is taken to be the externalname specified in the ALIAS clause if this is present. This is the exact name that will be used - the casing of letters will not be modified. If the ALIAS clause is missing the BASIC name without any type specifier is made lower case and an under-score (\_) character is inserted at the front. This is appropriate for linking with Lattice C routines.

### • See

FUNCTION, SUB, User Manual : Concepts chapter, Appendix F: Linking.

### • Example

```
DECLARE FUNCTION double(BYVAL i)
```

```
PRINT double(23)
```

```
FUNCTION double(BYVAL i)
double=i*2
END FUNCTION
```

```
DECLARE SUB External CDECL (BYVAL i%)
```

```
External 23
```

```
' This calls _external with a parameter of 23 on the stack
```

```
DECLARE FUNCTION Comp CDECL(BYVAL a!,BYVAL b!)
DECLARE SUB assem CDECL ALIAS "assembler" (BYVAL
para1&,para2)
DECLARE SUB no_paras CDECL ALIAS "ASSEM"
DECLARE FUNCTION cfunc& CDECL
```

## DECR statement

---

### • Syntax

DECR *numeric\_variable*

### • Effect

Subtracts one from the specified variable

### • Comments

*numeric\_variable* may be a simple numeric variable or a numeric array element of any type. If you wish to decrement a simple variable X it probably makes more sense stylistically to use  $X=X-1$  rather than DECR X, but if you have a complicated array expression as below DECR is the appropriate construct.

### • See

INCR.

### • Example

DECR A(B(I,j+5)) 'equivalent to  $A(B(I,j+5))=A(B(I,j+5))-1$

## DEF FN statement

### • Syntax

```
DEF FNfunction_name[(parameter_list)]=expression
```

or

```
DEF FNfunction_name[(parameter_list)]  
[LOCAL variable_list]  
[STATIC variable_list]  
[SHARED variable_list]
```

.

statements

.

[EXIT DEF]

[FNfunction\_name=expression]

.

END DEF

### • Effect

Define old style single- or multi-line functions with parameters.

### • Comments

In general we recommend that you use FUNCTIONS rather than this old style of DEF FN as FUNCTIONS can easily be turned into sub-programs and vice versa.

The *function\_name* must be a unique identifier in your program. The optional *parameter\_list* can contain any number of variable names separated by commas, each of which may be preceded by VARPTR to denote variable rather than value parameters (see the *User Manual : Concepts* chapter for more details of parameter types).

The *function\_name* must be a variable name and the type returned is determined by the same rules as for variables. Thus FNfred% returns an integer. FNjohn will normally return a single precision floating point value unless you have used a DEFTYPE statement affecting the letter J. Note that DEF FN function names must be preceded by FN.

Inside function definitions, variables are assumed to be global unless you use a STATIC or LOCAL statement to override this.

Function definitions may be recursive and functions may be called before they are declared. However function definitions can not be nested inside other function or sub-program definitions or within structured control statements.

FNfunction\_name=expression is used to return values from multi-line functions. You can have more than one such statement within a function definition.

EXIT DEF causes the current function to be exited immediately.

If you call multi-line functions from within expressions, be careful that you do not have problems with side effects. For example

```
DEF FNx  
  j=j+1  
  FNx=j  
END DEF  
j=4  
PRINT FNx+j
```

will probably not produce the same result as

```
j=4  
PRINT j+FNx
```

because the variable j is modified when the function is called.

It is also dangerous to use input/output statements within multi-line functions if you then call them from within another input/output statement. For example,

```
PRINT 54,FNsilly,j  
DEF FNsilly  
PRINT " Function silly is being called."  
FNsilly=1.4  
END DEF
```

is not sensible.

The best idea is either never to call functions in input/output or never to input/output within functions.

### • See

FUNCTION, *User Manual : Concepts* chapter.

### • Example

```
DEF FNaHex$(a&)="&H"+hex$(a&)  
DEF FNLongHex$(a&)  
STATIC a$  
a$=hex$(a&)  
IF LEN(a$)<8 THEN  
  a$=STRING$(8-LEN(a$),"0")+a$  
END IF  
FNLongHex$=a$  
END DEF  
PRINT FNaHex$(42),FNLongHex$(42)
```

## DEF SEG statement

### • Syntax

DEF SEG=numeric\_expression

### • Effect

This statement defines the mode of operation of PEEK and POKE.

### • Comments

In general we recommend the use of the PEEKB, PEEKW, PEEKL, SPEEKB, SPEEKW or SPEEKL functions and their POKE counterparts rather than the use of DEF SEG, PEEK and POKE which are provided for ST BASIC compatibility. The new style statements are faster and easier to understand.

If *numeric\_expression* in a DEF SEG statement is greater than 0 then 1 byte is PEEKed or POKEd and the value of the *numeric\_expression* is used as an offset of the address passed to PEEK or POKE.

If *numeric\_expression* = 0 and the address given to PEEK or POKE is of type single or integer then 2 bytes are PEEKed or POKEd.

If *numeric\_expression* = 0 and the address passed is of type long integer or double, then 4 bytes are PEEKed or POKEd.

The *numeric\_expression* is a long integer.

### • See

POKE, PEEKB, PEEKW, PEEKL, POKEB, POKEW, POKEL.

### • Example

```
REM $option G
DEF SEG=&HFFFF8200 'start of screen hardware
old=PEEK(3)         'protected byte-peek
POKE 3,old-1        'hack screen position
' the screen will have moved about a little at this point
INPUT a$            'wait for a key
POKE 3,old          'then restore screen
```

## DEFDBL, DEFINT, DEFLNG, DEFSNG, DEFSTR statements

### • Syntax

DEFDBL *letter\_range* [,*letter\_range*]...

DEFINT *letter\_range* [,*letter\_range*]...

DEFLNG *letter\_range* [,*letter\_range*]...

DEFSNG *letter\_range* [,*letter\_range*]...

DEFSTR *letter\_range* [,*letter\_range*]...

### • Effect

These statements declare variables and functions to be double precision, integer, long integer, single precision, or string.

### • Comments

The format of the *letter\_range* is *1st\_letter* [-*2nd\_letter*]. Allowable letters are upper- or lowercase. Any variable or function name beginning with a letter specified in a DEFtype statement defines the value of the variable: DBL for double precision, INT for integer, LNG for long integer, SNG for single precision, or STR for a string.

The type declaration characters #,%,&,!,\$ override any DEFtype statements.

**Note:** The variables I#, I%, I&, I!, and I\$ are all different variables with potentially different values.

### • See

User Manual : Concepts chapter.

### • Example

```
DEFINT a-z ' a good idea to have at the front of all
            ' programs so you only use floating point
            ' when needed and avoids many %s in names.
```

```
DEFINT i-n
DEFSNG a-h,o-z ' for old-fashioned FORTRAN programmers
                ' but confusing for everyone else.
```



## DIM statement

### • Syntax

```
DIM [SHARED]variable [ (subscripts...)]  
    [,variable [ (subscripts...)]]...
```

### • Effect

This statement defines the maximum values for array-variable subscripts and allocates the necessary storage.

### • Comments

The maximum subscript value for an unDIMmed array is 10. The maximum number of dimensions in an array is 31 (which would take up a minimum of 4 GigaBytes of memory if each index had more than one possible element). Subscript values can be integer or long integer expressions.

It is good programming practice to dimension all arrays even if you require 10 elements as this will enable you to switch off the array checks option.(A-) To ensure that you are not auto-dimensioning an array by mistake, use the Array Checks Warnings option (a+[])

The minimum value for each subscript may be set using the OPTION BASE statement.

If you use REM \$DYNAMIC in your program, all subsequent arrays that you dimension will be dynamic and can be modified with ERASE, REDIM, and REDIM PRESERVE. By default, or after using REM \$STATIC arrays will be static if the first dimension statement uses constants rather than more general expressions. Static arrays may not be ERASEd or REDIMensioned but they are accessed considerably faster.

When initially defined, the elements of a numeric array have the value 0. If a string array is defined, the elements have null values.

The DIM SHARED form is used to declare arrays and ordinary variables that can be accessed from within sub-programs without the need for a SHARED statement in each sub-program.

### • See

ERASE, REDIM, STATIC, LOCAL, *User Manual : Concepts chapter, Appendix B: Compiler Options.*

### • Example

```
DIM R(15)  
FOR I%=1 TO 15  
    R(I%)=I%  
    PRINT R(I%);  
NEXT I%
```

Result:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  
DIM SHARED debug_flag 'debug_flag can now be  
                        'accessed anywhere in the program.  
CONST maxsize=100  
varsize=100  
DIM a(100),b(maxsize),c(varsize),d(maxsize*2)  
' a & b are static arrays c & d are dynamic one so  
ERASE c  
REDIM d(varsize)  
'are fine but  
ERASE a  
REDIM b(varsize)  
'are illegal
```



## DO...LOOP statement

### • Syntax

```
DO [{WHILE|UNTIL} boolean_expression]  
.  
.  
[EXIT {DO|LOOP}]  
.  
{WEND|LOOP} [{WHILE|UNTIL} boolean_expression]
```

### • Effect

Repeats the statements within the DO...LOOP while the conditions are true (WHILE) or false (UNTIL).

### • Comments

The *boolean\_expressions* must be numeric expressions. If they evaluate to 0 this is taken as false; non-zero as true.

If an EXIT DO or EXIT LOOP statement is contained within the loop then control passes to the statement following the end of the loop. EXIT DO is the preferred form as this is used by Microsoft BASIC.

If DO...LOOPS are nested the inner most loop is exited. If you wish to exit an outer loop use the REPEAT loop construct, but note that unlike DO...LOOPS these are not portable to Microsoft BASIC.

here may a condition at both ends of the loop if desired or none at all; this is thus considerably more flexible than the WHILE...WEND loop.

### • See

WHILE, REPEAT.

## Examples

```
i=0  
DO  
    i=i+1  
    PRINT i;  
LOOP UNTIL i>=10
```

Result:

1 2 3 4 5 6 7 8 9 10

```
i=0  
DO  
    i:=i+1  
    IF i>10 THEN EXIT DO  
    PRINT i  
LOOP
```

Result:

1 2 3 4 5 6 7 8 9 10

## ELLIPSE statement

### • Syntax

```
ELLIPSE x_centre,y_centre,x_radius,y_radius  
        [,start_angle,end_angle]
```

### • Effect

Draws an ellipse or elliptical arc in the current window.

### • Comments

If the optional angle parameters are omitted, the command draws a hollow ellipse in the current line colour (the third parameter in the COLOR statement). The angle parameters are expressed in degrees times 10, and are measured anti-clockwise starting on the right side. Solid ellipses and arcs can be drawn using the PELLIPSE command. All the parameters are integers.

The x\_centre and y\_centre co-ordinates are based on the top left of the current window being (0,0) unlike the GEMVDI v\_ellipse command where (0,0) is the top left of the screen.

### • See

PELLIPSE, CIRCLE, PCIRCLE, v\_ellipse GEMVDI command.

### • Example

```
ELLIPSE 200,100,75,50  
ELLIPSE 100,100,50,100,225,1750
```

## END statement

### • Syntax

```
END  
END DEF  
END FUNCTION  
END IF  
END REPEAT name  
END SELECT  
END SUB
```

### • Effect

Ends a BASIC program, function definition, IF...THEN...ELSE block, SELECT statement, REPEAT loop or sub-program.

### • Comments

The END DEF statement indicates the end of a DEF FN.

END IF finishes an IF...THEN...ELSE block.

END FUNCTION indicates the end of a FUNCTION definition.

END REPEAT name ends a REPEAT loop.

END SELECT ends a SELECT CASE statement.

END SUB ends a BASIC sub-program; it is the counterpart of SUB. See the relevant statement for more details.

END alone terminates the entire BASIC program. The END statement may be put anywhere in the source.

### • See

EXIT.

### • Example

```
FUNCTION factorial&(BYVAL N%)  
    IF N% <= 1 THEN  
        factorial&=1  
    ELSE  
        factorial& = factorial&(N%-1)*N%  
    END IF  
END FUNCTION
```

## ENVIRON statement

### • Syntax

ENVIRON *string\_expression*

### • Effect

Modifies , creates or deletes an environment variable..

### • Comments

The string expression must be of the form:

envname=envvalue

or

envname envvalue

where the separator is a space. Either form will set the environment variable *envname* to have the value *envvalue*. If the *envvalue* is null or consists just of a single semi-colon then the environment variable will be deleted. This will effect the current program and any programs that are run within it; it won't change the environment of the parent process.

The environment string is stored in TOS memory, so if you are adding many strings to the environment ensure that you have left enough room for the operating system. Note that MicroSoft BASIC under MS-DOS does not re-allocate space for the environment in this way.

### • See

ENVIRON\$ function, COMMAND\$.

### • Example

```
ENVIRON "FRED=Hello"  
' sets the environment variable FRED to be HELLO
```

```
ENVIRON "CLINKWITH="  
'and  
ENVIRON "CLINKWITH=";  
'both remove any CLINKWITH environment variable.
```

## ENVIRON\$ function

### • Syntax

ENVIRON\$(*string\_expression*)

or

ENVIRON\$(*integer\_expression*)

### • Effect

Reads the value of an environment variable..

### • Comments

To find the current value of a given environment variable pass its name as a string .The string should *not* be terminated with an equals sign. A null string will be returned if the value can not be found. Note that this function is case sensitive so you should normally pass an upper case string and that just the value of the variable is returned without its name.

To find the names and values of the existing environment variables pass an integer *i* to ENVIRON\$ and it will return the *i*th name in the environment and its value. If there is no such variable the null string is returned. See the example below.

### • See

ENVIRON statement, COMMAND\$.

### • Example

```
PRINT ENVIRON$("PATH")  
' prints the current value of the PATH environment  
variable
```

```
' display all the current environment variables  
i=1  
DO  
  cur$=ENVIRON$(i)  
  IF cur$="" THEN EXIT DO  
  PRINT cur$  
  i=i+1  
LOOP
```

## EOF function

### • Syntax

EOF(*channel\_number*)

### • Effect

This function tests for the end-of-file condition of *channel\_number*.

### • Comments

-1 is returned if true; 0 if false. A common use for EOF is to test for the end-of-file when writing data to a sequential file. When writing to a random-access file, EOF returns true if the last GET was not able to read an entire record, due to an attempt to read beyond the end-of-file. The result returned is an integer.

### • See

OPEN, LOF, LOC.

### • Example

```
OPEN "TEST.DAT" FOR INPUT AS #1
Ctr=0
WHILE NOT EOF(1)
    junk$=INPUT$(1,#1)
    Ctr=Ctr+1
WEND
CLOSE #1
PRINT Ctr           'The length of the file in
                    'bytes can use LOF instead.
```

## ERASE statement

### • Syntax

ERASE *array\_name* [, *array\_name*]...

### • Effect

This statement de-allocates dynamic arrays previously defined with DIM.

### • Comments

An array once ERASEd can only be accessed again after it has been DIMmed. REDIM alone will ERASE and DIM an array together, whereas REDIM PRESERVE allows an array to be extended or truncated without losing its remaining data. This command can move existing arrays in memory, so extreme care should be taken if used within functions or sub-programs that are called with arrays or array elements as variable parameters.

Note that only dynamic arrays may be ERASEd.

See the *User Manual : Concepts* chapter for more details.

### • See

DIM, REDIM, *User Manual : Concepts* chapter.

### • Example

```
REM $dynamic
DIM Arr(15000)

' some processing with Arr()

ERASE Arr           ' free the space used by Arr
```

## ERL, ERR functions

### • Syntax

ERL  
ERR

### • Effect

These functions return error status of the current program.

### • Comments

When running an error handling routine, ERR will return the error code and ERL returns the line number in which the error occurred. Note that if the error occurred on a line without a line number, the most recent line number found in the program will be returned in ERL.

If no line number at all has been encountered ERL will return the physical line number where the error occurred. Note that this is the line number after any include files have been inserted by the compiler. This behaviour without line numbers is a HiSoft BASIC 2 extension.

### • See

ON ERROR, ERROR.

### • Examples

```
IF ERR=53 THEN
  PRINT "File not found error on or after line ";ERL
END IF
```

```
99 PRINT "FINISHED"
RETURN
I=0
PRINT 1/I
```

'After executing 1/I the value of ERL will be 99 even  
'though this line number has nothing to do with the code  
'executed.

## ERROR statement

### • Syntax

ERROR *integer\_expression*

### • Effect

This statement simulates the occurrence of a BASIC run time error.

### • Comments

ERROR is the counterpart to ERR in that *integer\_expression* is the code returned by ERR. The user can also create his own error codes by using error numbers not already used by BASIC.

### • See

ERL, ERR, ON ERROR.

### • Example

```
IF Tst <= 0 THEN ERROR 99      'An error code not reserved
                                'by BASIC
IF ERR = 99 THEN
  PRINT "Something has gone terribly wrong in SUB TEST"
END IF
```

## EXIT statement

### • Syntax

EXIT {DEF|DO|FUNCTION|IF|FOR|LOOP|SELECT|SUB|*identifier*}

### • Effect

Exits a function definition, BASIC sub-program or a structured statement.

### • Comments

EXIT DEF exits from a function definition whereas EXIT SUB exits from a sub-program. EXIT DO, EXIT IF, EXIT FOR, EXIT FUNCTION, EXIT LOOP and EXIT SELECT cause the corresponding structured statement to be exited immediately and execution continues with the statement after the corresponding END.

EXIT *identifier* causes the named REPEAT loop to finish prematurely.

These statements are used to leave routines (e.g. as part of an IF...THEN...END IF block). They do not define the end of the routines definitions; END DEF, END FUNCTION, END SUB and END REPEAT *identifier* are required to do this.

### • See

DO, IF, FOR, FUNCTION, LOOP, SELECT, SUB.

### • Example

```
SUB Fred
.
.
IF Var&=12654 THEN EXIT SUB
.
.
END SUB
FOR i=1 TO max
  IF a(i)=0 THEN EXIT FOR
NEXT i
IF i>max THEN
  PRINT "no zero elements"
ELSE
  PRINT "first zero element is";i
END IF
```

## EXP function

### • Syntax

EXP(*numeric\_expression*)

### • Effect

Calculates the exponential function of the *numeric\_expression*.

### • Comments

This function returns *e* to the power of the *numeric\_expression*. *e* is the base of natural logarithms. If the *numeric\_expression* is double precision then the result is calculated to double precision otherwise single precision is used.

### • See

LOG.

### • Example

```
PRINT EXP(1#)
```

Result:

2.718281828459046

## FEXISTS function

### • Syntax

FEXISTS(*filename*)

### • Effect

Determines whether or not a particular file exists.

### • Comments

*filename* should be a legal GEMDOS filename, together with a drive specifier and sub-directory if required. The function returns -1 if the file is found, or 0 if it is not.

### • See

OPEN.

### • Example

```
f$="TEST.DAT"
IF NOT FEXISTS(f$) THEN
  f$="B:\"+f$
  IF NOT FEXISTS(f$) THEN
    do_error "Cannot find data"
  END IF
END IF
OPEN f$ FOR INPUT AS #1
```

## FIELD statement

### • Syntax

FIELD [#]*channel\_number*, *field\_width* AS *string\_variable*...

### • Effect

This statement allocates space for variables in a random access buffer.

### • Comments

The *channel\_number* is the number which was used to OPEN the file. *field\_width* is the amount of space in the *string\_variable* to be allocated, in characters.

The total number of bytes allocated must not exceed the record length specified when opening the file; default record length is 128 bytes.

When a file is CLOSED, all FIELD definitions are lost. All fielded strings that BASIC associates with the file receive null values.

A FIELD statement does not cause data to be put into the buffer. It associates a specified amount of space within the random-access buffer with a variable.

**Note:** Do not use a fielded variable in an INPUT statement if it is to remain fielded. An INPUT or other assignment will cause the variable's pointer to refer to string space, not the random-access buffer. You should use LSET and RSET to assign to field variables.

### • See

OPEN, GET, PUT, LSET, RSET, *User Manual* : *Concepts* chapter.

### • Example

```
'The first definition allocates the 12 byte buffer as the
'country code, area code and phone number; the second
'allocates the same buffer entirely to the single
'variable of Wholenum$.
OPEN "JOHN.DAT" FOR INPUT AS #1 LEN=12
FIELD #1, 2 AS Ccode$, 3 AS Acode$, 7 AS Number$
FIELD #1, 12 AS Wholenum$
```



## FILES statement

### • Syntax

FILES [*file\_spec*]

### • Effect

This statement lists the names of files in the specified directory on a disk.

### • Comments

The *file\_spec* must be a string conforming to GEMDOS conventions and may contain a drive letter, pathname, and file name including wildcards. If *file\_spec* is omitted, the current directory of the current drive is listed.

The format if the directory is the same as that produced by Microsoft BASIC, except the free space remaining on the disk is not printed, because of the time taken by such a function under TOS versions prior to 1.04.

### • Example

```
FILES "A:\DEVPAC\*.S" 'This produces a list of all
                        'the files in the DEVPAC
                        'directory of the disk in
                        'drive A: which have the
                        'extension S.
```

## FILL statement

### • Syntax

FILL *x\_start*,*y\_start*

### • Effect

This statement fills enclosed shapes already drawn on the screen.

### • Comments

The pixel in the current output window specified by *x\_start* and *y\_start* is plotted and the pattern 'spreads out' to fill the whole of the shape. The current fill colour, style and index, as selected by the COLOR statement are used.

The *x\_start* and *y\_start* parameters are specified relative to the current window, unlike the corresponding GEMVDI command *v\_contourfill*.

### • See

*v\_contourfill*.

### • Example

```
COLOR 1,1,1,0,0
BAR 70,40,160,120      ' hollow box
ELLIPSE 150,100,70,50  ' and ellipse
COLOR 1,1,1,1,4
FILL 71,50
```

## FIX function

### • Syntax

FIX(*numeric\_expression*)

### • Effect

This function produces the truncated integer part of the *numeric\_expression*.

### • Comments

The difference between FIX and INT is that if the *numeric\_expression* is negative, FIX will return the first negative integer greater than itself, whereas INT returns the first negative integer less than the *numeric\_expression*. The type of the result will be the same as the type of *numeric\_expression*.

### • See

INT, CINT, CLNG, CDBL, CSNG.

### • Example

```
PRINT FIX(1.5)
PRINT FIX(-1.5)
```

Result:

```
1
-1
```

## FOR...NEXT statement

### • Syntax

```
FOR counter = start TO end [STEP increment]
```

```
.
```

```
NEXT [counter ][,counter]...
```

### • Effect

This statement executes a series of instructions in a loop a specified number of times.

### • Comments

|                  |  |
|------------------|--|
| <i>counter</i>   | is a variable used as the loop counter   |
| <i>start</i>     | is the value of <i>counter</i> at the beginning of the loop                        |
| <i>end</i>       | is the value of <i>counter</i> upon which the loop ceases to execute               |
| <i>increment</i> | is the value by which <i>counter</i> is incremented each time the loop is executed |

All four values may be either integers, long integers, single or double precision floats.

The program lines between FOR and NEXT are executed repeatedly. *counter* is adjusted by *increment* after execution of each iteration of the loop. *counter* is then compared with *end*. If *counter* is greater than *end*, execution of the program continues on the line after NEXT.

Note that *counter* is updated before the comparison is made.

Unless STEP *increment* is specified, the increment value is one. If *increment* is negative, *end* must be less than *start* otherwise the loop will not be executed.

If *start* is less than *end* and *increment* is positive then the loop isn't executed.

FOR...NEXT loops may be nested, provided that each loop has a unique *counter* variable. The NEXT statement for the inside loop must appear before the NEXT statement for the outside loop.

A NEXT statement of the format NEXT *x*, *y*, *z* is the same as

```
        NEXT x
      NEXT y
NEXT z
```

A NEXT statement without specifying *counter* will match the most recent FOR statement.

FOR...NEXT loops may finish prematurely via the use of EXIT or GOTO statements.

*Note:* The sum of *end* and *increment* (even of *increment* isn't explicitly specified, and is therefore one) must never exceed the range of *counter*'s variable type otherwise overflow results.

*Optimisation:* A FOR...NEXT loop executes fastest when *counter* is a integer and *start*, *end* and *increment* are constants. If any of the last three values must be variables, they should be made integers for maximum speed.

- **See**

WHILE, DO...LOOP.

- **Example**

```
DEFINT I
FOR I_ONE = 1 TO 5
  FOR I_TWO = 3 TO 11 STEP 2
    FOR I_THREE = 0 TO -15 STEP -1
      .
    .
  .
NEXT I_THREE, I_TWO, I_ONE
```

## FORMAT\$, FORMATL\$, FORMATS\$ and FORMATD\$ functions

- **Syntax**

```
FORMAT$(integer_expression,string_expression)
FORMATL$(long_expression,string_expression)
FORMATS$(single_expression,string_expression)
FORMATD$(double_expression,string_expression)
```

- **Effect**

These functions format numeric expressions in a similar manner to the PRINT USING and PRINT statements but they produce their result as a string.

- **Comments**

These functions can be used to convert numbers to strings for manipulation yourself rather than for direct output. If the *string\_expression* is null then the string produced consists of the characters that would be printed in a normal PRINT statement; otherwise *string\_expression* is used as a PRINT USING format string. See the PRINT USING statement for full details of this formatting.

These functions are particularly useful when displaying numbers in dialog boxes or when printing directly with GEMVDI.

- **See**

PRINT USING, PRINT, STR\$.

- **Example**

```
WRITE FORMAT$(12,""),FORMAT$(12,"###")
WRITE FORMAT$(1E10,""),FORMAT$(1E10,"##.###^")
WRITE FORMATD$(1E10,""),FORMATD$(1E10,"##.###^")
```

Result:

```
" 12 ", " 12"
" 1E+10 ", " 1.00E+10"
" 10000000000 ", " 1.00D+10"
```

## **FRE function**

### • **Syntax**

FRE(*numeric\_expression*)

FRE(*string\_expression*)

### • **Effect**

This function returns the size of free heap space or GEMDOS free memory.

### • **Comments**

FRE returns the following values:

For a *numeric\_expression* which is not equal to -1, the size of the free space on the BASIC heap is returned. For compatibility with other versions of HiSoft BASIC use zero or a positive value for this. For -1, the size of GEMDOS free memory is returned.

For a *string\_expression*, the size, in bytes, of contiguous free heap space after a garbage collect is returned.

The *heap* is the area of memory used by a BASIC program for storing string variables and arrays. It is used up as your program executes string expressions but the garbage thus created will be 'cleaned up' automatically by the garbage collector. Thus the value returned by FRE(0) becomes smaller and smaller until a garbage collect occurs when it will return to a much larger value.

### • **Example**

```
PRINT FRE(0)
PRINT FRE("")      'This is a null string
PRINT FRE(-1)
```

Result:

```
127534
154293
4096
```

## **FREEFILE function**

### • **Syntax**

FREEFILE

### • **Effect**

This function returns the first free file number as an integer.

### • **Comments**

FREEFILE can be used in sub-routines to obtain a free file channel number for file input/output so that you can proceed without worrying about which file numbers are already used by the program.

### • **See**

OPEN, CLOSE.

### • **Example**

```
'This routine displays a text file on the screen without
'interfering with any existing files in use
SUB ShowFile(filename$)
  STATIC myfile,temp$
  myfile=FREEFILE
  OPEN filename$ FOR INPUT AS #myfile
  WHILE NOT EOF(myfile)
    LINE INPUT #myfile,temp$
    PRINT temp$
  WEND
  CLOSE myfile
END SUB
```

## GET file I/O statement

### • Syntax

GET [#]channel\_number[,record\_number]

### • Effect

This statement reads a record from a random-access disk file into a random access file buffer.

### • Comments

The *channel\_number* is the number under which the file was opened. If *record\_number* is omitted, the next record after the last GET is read.

EOF is useful to use in conjunction with GET to check if GET was beyond the actual end of the file.

### • See

OPEN, FIELD, PUT, INPUT#, *User Manual : Concepts* chapter.

### • Example

```
'This routine prints the names and ages of the first 10
'people listed in AGES.DAT
DEFINT I
OPEN "AGES.DAT" FOR RANDOM AS #1
FIELD #1, 10 AS Fname$, 10 AS Lname$, 2 AS Age$
FOR I = 1 TO 10
    GET #1,I           'Read the first 10 entries in AGES
    PRINT Fname$;" ";Lname$;" is ";Age$;" years old."
NEXT I
CLOSE #1
```

## GET graphics statement

### • Syntax

GET(x1,y1)-(x2,y2),array\_name[(index\_expression...)]

### • Effect

This statement stores binary images from any part of the screen.

### • Comments

GET transfers a rectangular image defined by (x1,y1)-(x2,y2) into the array specified by *array\_name*. PUT is the counterpart statement; it transfers the image from the array to the screen.

(x1,y1)-(x2,y2) represent diagonally opposite corners of the rectangle on the screen.

Note that the co-ordinates are expressed relative to the top left of the screen, not the current window, unlike most other graphic commands.

The size (in bytes) of an array needed to store an image is

$$2 * p * (y2 - y1 + 1) * [(x2 - x1) \setminus 16 + 1] + 6$$

where p is the number of planes of screen memory, i.e. 1 for high resolution, 2 for medium and 4 for low. For example to store an image of (10,20)-(100,200) in low resolution would require  $2 * 4 * 181 * 6 + 6 = 8694$  bytes, or an integer array of 4347 elements.

The GET command has its own idea of the *current* screen; to change the current screen (e.g. for animation) you should update the long word at SYSTAB+50.

If an integer array p%() is used in a GET statement then

|       |   |
|-------|---|
| p%(0) | contains the width of the image             |
| p%(1) | contains the height of the image            |
| p%(2) | contains the number of planes in the image. |

Note that this form of array is compatible with HiSoft BASIC on the Amiga but is *not* the same as for HiSoft BASIC 1 on the ST.

Multi-dimensional arrays may be used in this statement, making it easy to switch between different images.

The following example shows how to get the entire screen in a resolution independent way that will work in any ST or TT mode or with any add-on screen for that matter.

### • See

PUT, SYSTAB, vro\_cpyfm VDI command.

## • Example

```
p=PEEKW(SYSTAB)      'planes
x1%=PEEKW(SYSTAB+40)  'workstation co-ordinates
y1%=PEEKW(SYSTAB+42)
x2%=PEEKW(SYSTAB+44)
y2%=PEEKW(SYSTAB+46)
DIM whole%(CLNG(p)*(y2-y1+1)*[(x2-x1)\16+1] + 2) 'just
GET (x1,y1)-(x2,y2),whole% 'get the whole screen

'display it 10 pixels down and to the right
PUT (10,10),whole%,PSET

DIM pics%(2,2503)    'enough for 16 colours
....
GET (0,0)-(99,99),a(1,0)
...
GET (0,0)-(99,99),a(2,0)
FOR i=1 TO 2          'display both
    PUT (0,0),a(i,0),PSET
    INPUT a$
NEXT i
```

## GETCOOKIE function

### • Syntax

GETCOOKIE(*string\_expression*,*long\_integer\_variable*)

### • Effect

Reads the value of a 'cookie' from the Cookie Jar. The value returned by the function is -1 if the cookie is found or 0 if it is not.

### • Comments

The Cookie Jar is a convention, introduced in STE TOS, whereby the system (and third party suppliers) can indicate the capabilities of an Atari TOS machine.

Each cookie has a 4 character name and a long integer value. Cookies beginning with \_ are reserved for Atari's system cookies and, at the time of writing, are as follows:

|      |   |
|------|---|
| _CPU | the bottom 2 digits of the main processor number (e.g. 0 for 68000, 30 for 68030)   |
| _VDO | the major/minor part number of the video shifter. At present the least significant word is always zero and the high word is one of<br>0 ST<br>1 STE<br>2 TT |
| _SND | This is bit oriented as follows:<br>bit 0 1 if ST style GI/Yamaha chip available<br>bit 1 1 if TT/STE style DMA sound available                             |
| _MCH | This is rather more vague and gives the machine type. As with _VDO this consists of a high word as follows:<br>0 520/1040 or Mega ST<br>1 STE<br>2 TT       |

The low word is normally zero except in the case of the MegaSTE where the low word is 16. Thus \_MCH is &h10010 for the MegaSTe.

Normally you should use the more specific cookies given above, in case some one has added a 68030 processor to an STE, for example.



\_SWI

The Mega STE and TTs have have internal configuration switches; this gives their value. However normally you should not access this directly; another cookie would be more appropriate.

\_FRB

'Fast RAM Buffer'. This is used on the TT to give the address of a 64K buffer in system (chip or ST) RAM that all ACSI devices performing DMA can use, when transfers to TT RAM are requested. It is not present if there is no fast RAM.

\_FPU

This gives an indication of any Floating Point Unit installed in the machine. As with \_VDO and \_MCH only the high word is used at the time of writing. Bit 0 is used to indicate if an I/O mapped 68881 is present (like Atari's SFP004 or on the MegaSTe). The following values of Bits 1 to 3 indicate

- 1 68881 or 68882 (unsure which)
- 2 definitely 68881
- 3 definitely 68882
- 4 68040 internal floating point support

Note that the table above does not give the usage of the individual bits but the value given by those 3 bits together.

Thus on a machine with a 68882 and no I/O mapped processor this cookie will have the value &h2000 or &h6000.

The reason for the 'unsure' status is that the BIOS detects whether a co-processor is present, but does not attempt to determine which one.

At the time of writing most Mega STs with I/O mapped co-processors do *not* have a cookie jar.

Although the cookie jar was introduced with STE TOS it can be retro-fitted to earlier STs

### • Example

```
IF GETCOOKIE("_CPU",cpu&) THEN
  PRINT "This machine has a ";68000+cpu&;" processor"
ELSE
  PRINT "This machine doesn't have a _CPU Cookie"
  PRINT "So its probably a 68000"
END IF
```

```
IF GETCOOKIE("_VDO",vdo&) THEN
  IF (vdo& AND &hFFFF0000)=&h20000 THEN
    'Now we can use the XBIOS TT video calls
    ....
  END IF
```

## GOSUB...RETURN statements

### • Syntax

```
GOSUB {line_number1|line_label1}
```

.

```
RETURN {line_number2|line_label2}
```

### • Effect

These statements branch to, and return from, a subroutine, respectively.

### • Comments

*line\_number1* or *line\_label1* identifies the first line of the subroutine to which GOSUB branches. You can also RETURN from a subroutine to *line\_number2* or *line\_label2* and not necessarily to the line after the GOSUB call. This is not generally recommended as it can lead to very unreadable code.

A subroutine may contain more than one RETURN statement. A RETURN without *line\_number2* or *line\_label2* branches back to the line after the GOSUB that called the subroutine.

*Note:* We suggest that you use CALLs to SUB...END sub-programs rather than GOSUB...RETURN. SUB...END routines are much more flexible in that parameters can be passed to them, they can support local variables.

HiSoft BASIC does not support GOSUB to a sub-program.

### • See

SUB, FUNCTION.

### • Example

```
I% = 0
PRINT "Main Loop"
GOSUB Loop1
Back In Main:
PRINT "Back in Main Loop"
I% = 1
GOSUB Loop2
PRINT "Done."
END
.
.
Loop1:
PRINT "In 1st subroutine"
RETURN Loop2
```



```

Loop2:
PRINT "In 2nd subroutine"
IF I% = 0 THEN
    RETURN Back_In_Main
ELSE
    RETURN
END IF

```

Result:

```

Main Loop
In 1st subroutine
In 2nd subroutine
Back in Main Loop
In 2nd subroutine
Done

```

## GOTO statement

### • Syntax

GOTO {*line\_number*|*line\_label*}

### • Effect

This statement causes program execution to unconditionally jump to the line specified.

### • Comments

GOTOs must *not* be used to enter or leave functions, subprograms or subroutines. GOTOs can be used within these structures if necessary.

*Note:* We suggest the use of structured control statements such as IF...THEN...ELSE and DO...LOOP for enhanced readability of the source and easier debugging. If you need to use a GOTO use a meaning label rather than a line number.

### • See

IF, DO, EXIT.

### • Example

```

GOTO Label
'The program never gets here
.
.
Label:
PRINT "The program resumes execution here"

```

## GOTOXY statement

---

- **Syntax**

GOTOXY *x\_character*,*y\_character*

- **Effect**

Move the text cursor to a particular location on the screen.

- **Comments**

The top-leftmost position on the screen or output window is (0,0).

*Note:* In order to be compatible with the first version of ST BASIC *x\_character* is doubled before the cursor is positioned when in high resolution.

It is recommended that the LOCATE statement is used instead, as it allows more accurate cursor positioning. Note that LOCATE has the parameters in the opposite order and that (1,1) is the top left.

- **See**

LOCATE, PRINT, POS, CSRLIN.

- **Example**

```
GOTOXY 0,21
INPUT "Enter name: ",n$
```

## HEX\$ function

---

- **Syntax**

HEX\$(*numeric\_expression*)

- **Effect**

This function returns a string that represents the hexadecimal value of the *numeric\_expression*.

- **Comments**

The *numeric\_expression* is rounded before being evaluated by HEX\$. If the parameter is an integer then the resulting string will be from 1 to 4 characters in length, and if it is a long integer then the result may be up to 8 characters in length.

- **See**

OCT\$, BIN\$.

- **Example**

```
'This routine prints the numbers 1 to 15 in hexadecimal
'notation
DEFINT I
FOR I=1 TO 15
    PRINT HEX$(I);
NEXT I
```

Result:

123456789ABCDEF

## IF...THEN...ELSE statement

### • Syntax

```
IF boolean_expression THEN  
    statement_1  
    [statement_2]
```

```
·  
·
```

```
[ELSEIF boolean_expression THEN  
    statement_3  
    [statement_4]]
```

```
·  
·
```

```
[ELSE  
    statement_5  
    [statement_6]]
```

```
·  
·
```

```
END IF
```

or

```
IF boolean_expression THEN statement_1 [ELSE statement_2]
```

### • Effect

This statement block allows conditional execution or branching, based on the evaluation of a Boolean expression.

### • Comments

In the single-line variation of IF...THEN...ELSE, *statement\_1* is executed if *boolean\_expression* evaluates to true. If *boolean\_expression* is false, then *statement\_2* is executed. If ELSE is not present and *boolean\_expression* is false, the program resumes execution on the line after the IF...THEN statement.

If there are any statements after the THEN the single line form is used and no END IF is required.

Thus the following is illegal:

```
IF a THEN PRINT "line 1"  
    PRINT "line 2"  
END IF
```

whereas

```
IF a THEN  
    PRINT "line 1"  
    PRINT "line 2"  
END IF
```

is correct.

### • See

SELECT.

### • Example

```
S!=RND  
IF S!<0.5 THEN  
    PRINT "S! is smaller than 0.5"  
ELSEIF S!>0.5 THEN  
    PRINT "S! is larger than 0.5"  
ELSE  
    PRINT "S! is equal to 0.5"  
END IF
```

## INCR statement

- **Syntax**

INCR *numeric-variable*

- **Effect**

This statement adds one to the *numeric\_variable*.

- **Comments**

The *numeric\_variable* may be a simple or array variable of any numeric type, the value of which is incremented by one.

- **See**

DECR.

- **Example**

INCR a(i,j) 'equivalent to a(i,j)=a(i,j)+1

## INKEY\$ function

- **Syntax**

INKEY\$

- **Effect**

This function reads a character from the keyboard without echo, if one is available.

- **Comments**

INKEY\$ returns either a null, one- or two- byte string containing the character read from the keyboard.

A null string result (i.e. " ") means no key was pressed.

A one-character string means a 'normal' key was pressed, and contains the ASCII value of it.

A two-character string means a non-ASCII key was pressed, whose scan code can be found in the second character - the first character will be CHR\$(0). Keys such as the function keys, cursor keys, and Alt-combinations are non-ASCII.

- **See**

evnt\_keybd AES function.

- **Example**

'This block provides a quicker way of processing user input, if only one character is needed.

PRINT "Enter your choice (1 or 2)"

DO

    a\$=INKEY\$

    SELECT CASE a\$

        CASE ="1"

            CALL One: EXIT DO

        CASE ="2"

            CALL Two: EXIT DO

    END SELECT

LOOP

## INP function

### • Syntax

INP(*BIOS\_handle*)

### • Effect

This function returns a byte value from the device specified by *BIOS\_handle*.

### • Comments

INP is the complement of the OUT function. It waits for a byte from the specified device then returns its value.

*BIOS\_handle* is defined as:

|   |                |
|---|----------------|
| 0 | parallel port  |
| 1 | serial port    |
| 2 | keyboard       |
| 3 | MIDI interface |

The status of the device can be polled by using a negative value for *BIOS\_handle*. A returned value of 0 indicates no character is available; -1 indicates that a character is available. The argument and result are both integers.

If the keyboard is read (2) and the bottom eight bits would be zero, then the value returned is the high word of the result with bit 7 set. This strange behaviour is provided for compatibility with ST BASIC which used this 'fudge' to return information about the function keys etc whilst still using a short integer. In general we recommend calling the BIOS BConin or AES evnt\_keybd routines directly for this.

### • See

OUT, evnt\_keybd AES function, conin& BIOS function.

### • Example

```
Listen:
IF INP(-3) THEN
  CALL MIDI_rd
ELSE GOTO Listen
.
.
SUB MIDI_rd
  Char%=INP(3)
.
.
```

## INPUT statement

### • Syntax

INPUT [;][*"prompt"*{;|,}] *variable\_list*

### • Effect

This statement prompts the user for input which is assigned to *variable\_list*.

### • Comments

INPUT causes program execution to halt and await user data. If a *prompt* is included, INPUT will first print it and then a question mark. If the question mark is to be suppressed, a comma should be put after *prompt*. If the cursor is to stay on the same line even after user presses enter (i.e. not echo the CR-LF), a semi-colon should be put immediately after the INPUT statement.

### • See

INKEY\$, evnt\_keybd, INP, PRINT.

### • Example

```
INPUT "The square of",Val%
PRINT "is";(Val%*Val%)
```

Result:

The square of 2 is 4

## INPUT# statement

- **Syntax**

INPUT #*channel\_number*,*variable\_list*

- **Effect**

This statement reads data from the device or file specified by *channel\_number* and assigns it to *variable\_list*.

- **Comments**

The *channel\_number* is the number specified when OPENing the file.

The data read must be numeric for numeric variables and leading carriage returns, line feeds, and spaces are ignored; the first character that is not one of the three preceding types is considered to be the beginning of a number. A number is terminated by space, carriage-return or a line feeds. If end-of-file is reached while a value is being read, the variable is terminated.

- **See**

BLOAD, INPUT\$, PRINT#, Fread (GEMDOS call).

- **Example**

```
INPUT #1, Str$, Int%, Lng_int&, Sing!, Dub_float#
```

## INPUT\$ statement

- **Syntax**

INPUT\$(*n*[, [#]*channel\_number*])

- **Effect**

This statement reads *n* characters from the specified channel.

- **Comments**

If *channel\_number* is omitted, the characters are read from the keyboard without echo. This can be used for input similar to INKEY\$, but with a fixed amount of characters to be processed from the keyboard. *n* may be an integer or long integer. Thus assuming there is sufficient memory available an entire file can be read with one use of this function.

- **See**

BLOAD, INPUT, PRINT#, Fread (GEMDOS call).

- **Example**

```
'This program checks if DATA.PRG has the valid GEMDOS
'program header beginning.
OPEN "A.PRG" FOR INPUT AS #1
first%=CVI(INPUT$(2,#1))
CLOSE #1
IF first%=&H601A
    PRINT "This file is executable."
ELSE
    PRINT "This is not a valid GEMDOS program file!"
END IF
```

## INSTR function

### • Syntax

INSTR([start],1st\_string,2nd\_string)

### • Effect

This function searches backwards for the occurrence of *2nd\_string* in *1st\_string*.

### • Comments

*start* is an optional offset for the beginning of the search within *1st\_string*. *start* may be either an integer or a long integer. The strings may be either string-variables, string-literals, or string expressions. The value returned is the position of the first character of the string that has been found.

If *2nd\_string* is found in *1st\_string* the location where it was found is returned.

If *start* is a value larger than the length of *1st\_string*, 0 is returned.

If the *1st\_string* is a null string, 0 is returned.

If *2nd\_string* cannot be found, 0 is returned.

If *2nd\_string* is a null string, *start* is returned; if *start* was not specified, 1 is returned.

*Note:* As strings in HiSoft BASIC have no length limits this function returns a long integer.

### • See

RINSTR.

### • Example

'This routine parses the command line for a space to find the first parameter passed to the program.

```
I%=INSTR(COMMAND$, " ")
```

```
IF I%=0 THEN I%=LEN(COMMAND$)
```

```
PRINT "The 1st parameter passed is: ";LEFT$(COMMAND$,I%)
```

## INT function

### • Syntax

INT(numeric\_expression)

### • Effect

This function returns the largest integer less than or equal to the *numeric\_expression*.

### • Comments

The value returned is the same type as *numeric\_expression*.

### • See

FIX, CINT, CDBL, CSNG, CDBL.

### • Example

'This example illustrates the differences between the functions CINT, FIX and INT.

```
PRINT CINT(1.5),CINT(-1.5)
```

```
PRINT FIX(1.5),FIX(-1.5)
```

```
PRINT INT(1.5),INT(-1.5)
```

Result:

|   |    |
|---|----|
| 2 | -2 |
| 1 | -1 |
| 1 | -2 |



## KILL statement

---

### • Syntax

KILL *file\_spec*

### • Effect

All files fitting the description of *file\_spec* are deleted from disk.

### • Comments

The *file\_spec* must conform to the GEMDOS format. KILL will not delete directories from disk, only their contents; use RMDIR to delete the actual directory.

### • See

Fdelete GEMDOS function.

### • Example

```
KILL "C:\BASIC\*.BAS" 'This line deletes all files
                        'with the extension BAS in the
                        'sub-directory BASIC on drive
                        'C: This is probably not what
                        'you want!
```

## LBOUND function

---

### • Syntax

LBOUND(*array*[,*dimension*])

### • Effect

This function returns the smallest available subscript of the specified array and optionally of a specific dimension.

### • Comments

A common use of LBOUND is to determine the size of an array, together with UBOUND. *array* is the array to be checked, whereas *dimension* is the number of the dimension to be checked.

In fact because OPTION BASE is used for all the subscripts of the array they will all have the same lower bound. However it may be possible to explicitly set the lower bound in the future.

### • See

UBOUND, OPTION BASE.

### • Example

```
DIM Levels(X,Y,Z)
.
PRINT LBOUND (Levels,2) 'This will return the lowest
                        'bound for the Y dimension
                        'which is 0 unless otherwise
                        'defined with OPTION BASE
```

## LCASE\$ function

---

- **Syntax**

LCASE\$(*string\_expression*)

- **Effect**

This function returns a string with every alphabetic character in lower case.

- **Comments**

Any characters in the range 'A' to 'Z' inclusive are converted to 'a' to 'z' as required. All other characters are left alone including accented letters.

- **See**

UCASE\$, ASCII table.

- **Example**

```
PRINT "Do you wish to format your hard disk? Y/N";
DO a$=INKEY$
UNTIL a$<>" "
IF LCASE(a$)="y" THEN
    do_format "C:"
ELSE
    PRINT "Very wise"
END IF
```

## LEFT\$ function

---

- **Syntax**

LEFT\$(*string\_expression*,*n*)

- **Effect**

This function returns a string made up of the leftmost *n* characters of *string\_expression*.

- **Comments**

*n* is a long integer argument as there is no string length limit in HiSoft BASIC. If *n* = 0 then a null string is returned. If *n* is greater than the number of characters in *string\_expression*, the entire string is returned. *string\_expression* may be a string variable, a string literal or a string constant.

- **See**

MID\$, RIGHT\$.

- **Example**

```
String$="HiSoft BASIC"
PRINT LEFT$(String$,6)
```

Result:

HiSoft

## LEN function

---

- **Syntax**

LEN(*string\_expression*)

- **Effect**

This function returns the number of characters in *string\_expression*.

- **Comments**

*string\_expression* may be a string variable, a string literal or a string constant. This can return numbers greater than 32767 so be careful if assigning to integers - use long integers instead.

- **Example**

```
Another_String$="DevpacST by HiSoft"  
PRINT LEN(Another_String$)
```

Result:

18

## LET statement

---

- **Syntax**

[LET] *variable* = *expression*

- **Effect**

Assigns *variable* the value of *expression*.

- **Comments**

LET is entirely optional and unnecessary; the equals sign is sufficient to assign values to variables. When assigning to a FIELDed variable you should use LSET or RSET.

- **Example**

```
LET profit=income-sales-tax
```

is the same as

```
profit=income-sales-tax
```

## LIBRARY statement

---

- **Syntax**

LIBRARY *library\_name*[,*another\_library\_name*]...

- **Effect**

This statement defines which libraries are to be used by the program.

- **Comments**

Libraries are additional functions and procedures written in assembly language but callable from HiSoft BASIC programs. The specified libraries must exist within the current HBASIC.LIB file. Library routines are called in exactly the same way as BASIC sub-programs and functions..

For details of the supplied libraries see *Chapter 2*. For information on writing your own libraries and general use of assembly language and C routines see *Appendix F*.

- **See**

DECLARE.

- **Example**

```
LIBRARY "xbios"  
BLOAD "TOOTH.PIC",logbase&
```

## LINEF statement

---

- **Syntax**

LINEF *x\_start*,*y\_start*,*x\_end*,*y\_end*

- **Effect**

Draws a line between two points in the current window.

- **Comments**

Both co-ordinates are specified relative to the top left of the window. The colour of the line can be specified using the COLOR statement.

Note that the co-ordinates are given relative to the current window, unlike the corresponding GEMVDI *v\_pline* command.

- **See**

BAR, COLOR, *v\_pline* GEMVDI routine.

- **Example**

```
FOR i=0 TO 300 STEP 3  
    LINEF i,0,300-i,200  
NEXT i  
FOR i=0 TO 200 STEP 3  
    LINEF 0,i,300,200-i  
NEXT i
```

## LINE INPUT statement

### • Syntax

LINE INPUT[;] ["prompt";] *string\_variable*

### • Effect

This statement assigns an entire line of input to *string\_variable* while ignoring delimiters (such as commas).

### • Comments

The *prompt* is printed before input is awaited. A question mark will not be displayed unless it is part of *prompt*. If LINE INPUT is immediately followed by a semi-colon, the CR-LF marking the end of the input line will not be echoed to the screen.

LINE INPUT is more useful if you need to enter strings with commas, quotation marks or spaces in them.

### • See

INPUT, READ, PRINT, WRITE.

### • Example

```
LINE INPUT "Enter command:",Com$
```

## LINE INPUT# statement

### • Syntax

LINE INPUT #*channel\_number*,*string\_variable*

### • Effect

This statement reads a sequence of characters terminated by a CR-LF from the device or file specified by *channel\_number* and assigns it to *string\_variable*.

### • Comments

This statement will read a sequence up to, and including, a CR-LF, and return. A subsequent LINE INPUT# will begin reading the second sequence after the CR-LF.

The LINE INPUT# statement allocates a buffer when reading a line. By default this is 1024 bytes enabling you to read lines up to 1K in length. If you wish to increase this you can change the long integer pointed to by SYSTAB+68. In general it is best to read long strings using INPUT\$ as this is much more efficient.

### • Example

'This routine reads a file in which 5 records that are  
'delimited by CR-LFs are kept.

```
DIM Record$(5)
OPEN "BASE.LST" FOR INPUT AS #1
I%=0
WHILE NOT EOF(1)
    I%=I%+1
    LINE INPUT #1,Record$(I%)
WEND
CLOSE #1
```

POKEL SYSTAB+68,2048 'increase the buffer to 2048  
bytes.

## LOC function

### • Syntax

LOC(channel\_number)

### • Effect

This function returns the program's current position within an OPENed file.

### • Comments

The value returned is a long integer. When used with random-access files, LOC returns the number of the last record read or written to the file. For sequential files opened for OUTPUT, INPUT or APPEND the value returned is the number of bytes written or read divided by 128.

### • See

LOF, fseek GEMDOS call.

### • Example

```
SUB WriteBlock
STATIC remember%
remember%=LOC(2)      'get current position
LSET a$=MKIS(remember%)
PUT #2,1              'use record 1 to hold last
                        'updated record number
PUT #2,remember       'and write actual record
END SUB
```

## LOCAL statement

### • Syntax

LOCAL variable\_list

### • Effect

This statement declares variables as local to function definitions and sub-programs and creates a new variable each time the function or sub-program is left and re-entered.

### • Comments

This statement can be used only within function definitions and sub-programs.

Variables in DEF FN function definitions usually are global or STATIC in sub-programs and FUNCTIONS. The LOCAL statement ensures that a new variable is created every time a recursive sub-program or function is called. See the *User Manual : Concepts* chapter for more details. Note that the LOCAL statement for arrays also dimensions the array, there is no need for a separate DIM statement.

### • Example

```
test=10
PRINT FNeasy,test
DEF FNeasy
STATIC test           'without this the global
test=5                'test would be changed here
FNeasy=1/test
END SUB
```

## LOCATE statement

### • Syntax

LOCATE *row* [, *column* [, *cursor*]]

### • Effect

This statement puts the cursor on *column* and *row* specified; it can also enable or disable the cursor.

### • Comments

If any parameter is omitted, LOCATE has no effect on that particular value.

*column* is the column number of the screen where the cursor is to be placed..

*row* is the row number on the screen where the cursor is to be placed.

If *cursor* is set to 0, the cursor is disabled; *cursor* set to 1 enables it.

The cursor is a small black rectangle, usually flashing when in a TOS program. For maximum speed the BASIC I/O system disables the cursor during character output. A cursor will always be shown when entering text via an INPUT statement regardless of its current setting.

If LOCATE is used within a window the range of *row* and *column* is determined by the current window size.

### • See

POS, CSRLIN.

### • Example

```
LOCATE 1,1,0 'this line moves the cursor to
              'the upper left corner of the
              'screen and disables the cursor.
```

## LOF function

### • Syntax

LOF(*channel\_number*)

### • Effect

This function returns the length of the file specified by *channel\_number*.

### • Comments

The value returned for the open file is a long integer.

### • See

LOC, Fseek GEMDOS call.

### • Example

```
'This routine opens a file, checks its length, allocates
' enough memory, and reads it in its entirety.
OPEN "LOTS.A.INF" FOR INPUT AS #1
Length=&=LOF(1)
FileBuf$=INPUT$(Length,&,#1)
CLOSE #1
```



## LOG, LOG10, LOG2 functions

---

### • Syntax

LOG(*numeric\_expression*)  
LOG10(*numeric\_expression*)  
LOG2(*numeric\_expression*)

### • Effect

LOG returns the natural logarithm (base *e*) of *numeric\_expression*.

LOG10 returns the logarithm base 10 of *numeric\_expression*.

LOG2 returns the logarithm base 2 of *numeric\_expression*.

### • Comments

*numeric\_expression* must be greater than 0.

*e* is approximately 2.718281828459046

All logarithmic functions are evaluated with single-precision accuracy by default. If the *numeric\_expression* is a double-precision number, LOGtype will be calculated in double-precision.

### • See

EXP, ^ operator in the *User Manual : Concepts* chapter.

### • Example

```
PRINT LOG10(10),LOG10(5)
```

## LPOS function

---

### • Syntax

LPOS(*argument*)

### • Effect

This function returns the number of characters printed via LPRINT since the last CR/LF..

### • Comments

The actual physical position of the printer head is not necessarily returned, as tabs are not expanded. The value returned is the position within the printer buffer starting at 1.

*argument* is a dummy parameter.

### • See

POS, LPRINT.

### • Example

```
'This routine continues printing strings until the  
'printer head reaches the 50th column  
WHILE LPOS(0)<50  
    I%=I%+1  
    LPRINT String$(I%)  
WEND
```

## **LPRINT, LPRINT USING statements**

---

- **Syntax**

LPRINT [expression\_list][{;|,}]

LPRINT USING format\_string; expression\_list [{;|,}]

- **Effect**

These statements print data through the current printer port.

- **Comments**

LPRINT and LPRINT USING are analogous to PRINT and PRINT USING, except that the data output is via the current printer port. Normally this is the parallel port but this can be changed using the Install Printer desk accessory, supplied on the Atari system disk or the Atari Printer CPX.

- **See**

PRINT, LPRINT USING, PCOPY.

- **Example**

```
LPRINT CHR$(12);    'This line send a form-feed to
                    'the device connected to the
                    'printer port.
```

## **LSET statement**

---

- **Syntax**

LSET string\_variable = string\_expression

- **Effect**

Left-justify a string variable, normally used for FIELDed variables.

- **Comments**

LSET left-justifies a string variable by padding with spaces on the right up to its length. This is normally used for FIELDed string variables, but can be used with ordinary string variables for formatting output. RSET is similar except that it right-justifies.

- **See**

FIELD, RSET, PUT.

- **Example**

```
FIELD #2,20 AS a$
LSET a$=GetData$(10)
PUT #2
```

## **LTRIM\$ function**

---

- **Syntax**

LTRIM\$(string\_expression)

- **Effect**

This function returns its string parameter with any leading spaces removed.

- **Comments**

LTRIM\$ trims spaces from the left. of a string This can be used to remove spaces at the beginning of user input.

- **See**

RTRIM.

- **Examples**

```
commandline$=LTRIM$(COMMAND$)
' commandline$ won't have spaces at the front
```

```
PRINT LTRIM$("  HiSoft")
PRINT LTRIM$("BASIC")
```

Result:

```
HiSoft
BASIC
```

## **MAX function**

---

- **Syntax**

MAX(expression,expression [,expression]...)

- **Effect**

Finds the maximum of two or more values..

- **Comments**

MAX finds the largest of any number of expressions. The expressions may be of any numeric type or may all be strings.

- **Example**

```
INPUT "Enter three names";a$,b$,c$
PRINT "The first is ";MIN(a$,b$,c$)
PRINT "The last is ";MAX(a$,b$,c$)
```

## MID\$ function

- **Syntax**

MID\$(string\_expression,n[,length])

- **Effect**

This statement returns *length* characters from *string\_expression* starting at the *n*th character.

- **Comments**

If there are fewer than *length* characters in *string\_expression*, or if there are fewer than *length* characters to the right of the *n*th character, all characters following the *n*th character are returned. If there are less than *n* characters in *string\_expression*, a null string is returned.

- **See**

RIGHT\$, LEFT\$, SADD.

- **Example**

```
Sentence$="This is HiSoft BASIC"  
PRINT MID$(Sentence$,9,6)
```

Result:

HiSoft

## MID\$ statement

- **Syntax**

MID\$(string\_variable,n[,length])=string\_expression

- **Effect**

Modifies part of a string variable.

- **Comments**

The characters of *string\_variable* starting at position *n* are modified to be the characters of the *string\_expression*. If *length* is specified then only this number of characters are replaced; otherwise the whole of *string\_expression* is used.

MID\$ cannot be used to change the length a string and as a result is more efficient than an equivalent assignment statement. The *string\_variable* may be a simple string variable or an element of a string array. Both *n* and *length* may be integers or long integers.

- **See**

LSET, RSET.

- **Example**

```
Sentence$="This is HiSoft BASIC"  
MID$(Sentence$,1)=" It was"  
PRINT Sentence$
```

Result:

It was HiSoft BASIC

## MIN function

---

- **Syntax**

MIN(*expression*,*expression* [,*expression*]...)

- **Effect**

Finds the minimum of two or more values..

- **Comments**

MIN finds the least of any number of expressions. The expressions may be of any numeric type or may all be strings.

- **Example**

```
INPUT "Enter three values";a,b,c
PRINT "The smallest is ";MIN(a,b,c)
```

## MKDIR statement

---

- **Syntax**

MKDIR *pathname*

- **Effect**

This statement creates the sub-directory as specified by *pathname*.

- **Comments**

*pathname* must conform to the GEMDOS format conventions.

MKDIR can also be used with relative as well as absolute *pathnames*. For a detailed explanation of this, please refer to CHDIR.

- **See**

CHDIR, CURDIR\$, RMDIR, Dcreate GEMDOS call.

- **Example**

```
MKDIR "\ONE"      'an absolute pathname
CHDIR "\ONE"
MKDIR "TWO"       'this is a relative pathname;
                  'it creates \ONE\TWO
```

## MKI\$,MKL\$,MKSS\$,MKD\$ functions

### • Syntax

MKI\$(integer\_expression)  
MKL\$(long\_integer\_expression)  
MKSS\$(single\_precision\_expression)  
MKD\$(double\_precision\_expression)

### • Effect

These functions convert numeric data of the *expressions* into strings.

### • Comments

These functions take a numeric value and store it in a string variable. There is no conversion into human readable form. These functions are often used before outputting numeric data via LSET and RSET; both of these are only able to process strings. It is vital to note that these functions are not interchangeable with STR\$. A numeric value processed by a MKtype\$ function and then printed will be a binary representation of the number.

### • See

CVI, CVD, CVS, CVL, FIX, INT, STR\$.

### • Example

```
OPEN "FLD.DAT" AS #1
FIELD #1, 4 AS Lint$      'defines a field
Longint%=65536
LSET Lint%=MKL$(Longint%) 'puts the string value into
                           'the record
PUT #1,1                  'record is written to file
CLOSE #1
```

## MOUSE function

### • Syntax

MOUSE(attribute)

### • Effect

Read the current position of the mouse and the status of the buttons and keyboard shift keys.

### • Comments

The value returned is an integer, the interpretation of which depends on the *attribute*:

|   |   |
|---|---|
| 0 | X position of mouse                                   |
| 1 | Y position of mouse                                   |
| 2 | button state, 0=none, 1=left, 2=right, 3=both pressed |
| 3 | keyboard shift state                                  |
| 4 | Xposition of STE paddle 0                             |
| 5 | Y position of STE paddle                              |
| 6 | Xposition of STE paddle 1                             |
| 7 | Y position of STE paddle 1                            |
| 8 | Xposition of STE light pen in screen pixels           |
| 9 | Y position of STE light pen in screen pixels          |

Note that the mouse X and Y positions are relative to the screen top-left, not to any windows.

The keyboard shift state is returned as the sum of the following numbers:

|   |             |
|---|-------------|
| 1 | right Shift |
| 2 | left Shift  |
| 4 | Control key |
| 8 | Alt key     |

For example if both shift keys were held down the result of MOUSE(3) would be 3.

Note that the STE routines require the STE hardware - if it isn't there the machine will probably crash.

### • See

vq\_mouse & vq\_key\_s GEMVDI calls.

## • Example

```
WINDOW FULLW
CLS
DO      'plot points with left button
  IF MOUSE(2)=1 THEN PSET(MOUSE(0),MOUSE(1))
LOOP UNTIL MOUSE(3)=8  'until Alt key held down
```

## MOUSE statement

---

### • Syntax

MOUSE *mouse\_type*

### • Effect

Set the mouse to a pre-defined form or remove it from the screen.

### • Comments

If *mouse\_type* is -1 then the mouse is hidden from the screen, useful for example just before BLOADing a screen image or to increase the speed of screen output.

If *mouse\_type* is not -1 then the appearance of the mouse is changed to one of the following:

|   |                           |
|---|---------------------------|
| 0 | normal arrow              |
| 1 | vertical bar-like cursor  |
| 2 | 'busy' bee                |
| 3 | hand with pointing finger |
| 4 | outstretched hand         |
| 5 | thin cross                |
| 6 | thick cross               |
| 7 | outlined cross            |

### • See

graf\_mouse AES call.

### • Example

```
MOUSE -1
load_picture
MOUSE 4
drag_picture
```



## NAME statement

---

- **Syntax**

NAME *old\_filename* AS *new\_filename*

- **Effect**

This statement renames the file specified by *old\_filename* to *new\_filename*.

- **Comments**

Both file specifications must conform to the GEMDOS format. Unlike Microsoft BASIC under MS-DOS you cannot move files between directories by renaming them. Both arguments are string expressions.

- **See**

KILL, Frename GEMDOS call.

- **Example**

NAME "AFILE.BAS" AS "ANOTHER.BAS"

## OCT\$ function

---

- **Syntax**

OCT\$(*numeric\_expression*)

- **Effect**

This function returns a string which is the octal representation of *numeric\_expression*.

- **Comments**

OCT\$ returns the representation of the integer part of *numeric\_expression*. If the expression is an integer then the resulting string will be from 1 to 6 characters in length, but if it is a long integer then it can be up to 11 characters long.

- **See**

BIN\$, HEX\$

- **Example**

PRINT OCT\$(8).

Result:

10

## ON ERROR statement

### • Syntax

ON ERROR GOTO {*linenumber*|*linelabel*}

### • Effect

Enable error handling and specify error handling routine.

### • Comments

This command allows you to trap run-time errors such as overflow or disk full and pass control to a specific BASIC line if any errors occur. The error handler *must* be at the main level of your program - it cannot be within a sub-program or function.

If a line number of 0 is specified, any error handling is disabled and subsequent errors will abort program execution in the normal way. If ON ERROR GOTO 0 is specified within an error handler it will cause the original error message to be printed and program execution will be halted.

If you wish program execution to continue after an error you should use the RESUME statement.

Use of ON ERROR in a program causes a larger program size and slightly slower execution speed due to the saving of extra information while a program runs.

### • See

RESUME, ERL, ERR.

### • Example

```
ON ERROR GOTO handler
OPEN "DATA.INF" FOR INPUT AS #2
INPUT LINE #2,a$
CLOSE #2
PRINT "Data is ";a$ : STOP
handler:
IF ERR=53 THEN
    PRINT "Error - File DATA.INF not found" : STOP
ELSE
    ON ERROR GOTO 0 'another error so report
END IF
```

## ON...GOSUB statement

### • Syntax

ON *n* GOSUB {*linenumber*|*linelabel*}  
[, {*linenumber*|*linelabel*...}]...

### • Effect

This statement calls one of a list of subroutines depending on the value of a parameter.

### • Comments

If *n* has a fractional part it will be rounded to an integer.

*n* determines which *label* is jumped to. If *n* is 1 then the first subroutine is GOSUBed, else if *n* is 2 then the second label is the the subroutine that is executed, and so on for all the given subroutines.

If *n* is less than 1 or greater than the number of subroutines available, program execution continues after the ON...GOSUB statement.

It is normally best to use a SELECT CASE statement and sub-program calls rather than ON...GOSUB.

### • See

SELECT...CASE, ON...GOTO.

### • Example

```
ON Number% GOSUB One_routine, Two_routine, Three_routine
'If Number% = 3 then Three-routine will be executed.
```

## ON...GOTO statement

### • Syntax

ON *n* GOTO {*linenumber*|*label*} [, {*linenumber*|*label*}]...

### • Effect

This statement causes program execution to branch to one of a list of program lines depending on the value of a parameter.

### • Comments

This statement differs from ON...GOSUB in that program execution does not return to the line after this statement.

If *n* has a fractional part then it will be rounded to an integer.

*n* determines which label is jumped to. If *n* is greater than the number of labels specified, or if *n* is less than 1, program execution continues after the ON...GOTO statement.

We recommend using the SELECT CASE statement in preference to ON...GOTO.

### • See

SELECT...CASE, ON...GOSUB.

### • Example

```
ON Number% GOTO One_branch, Two_branch, Three_branch
'If Number% = 1 then program execution will continue at
'One_branch.
```

## OPEN statement

### • Syntax

OPEN *file\_spec* [FOR *mode*] AS [#]*channel\_num*  
[LEN=*record\_size*]

or

OPEN *mode\_string*, [#]*channel\_num*, *file\_spec* [,*record\_size*]

### • Effect

This statement prepares a file for reading or writing.

### • Comments

*file\_spec* must conform to GEMDOS specifications.

*mode* may be one of the following:

|        |  |
|--------|--|
| APPEND | Specifies a sequential file which is to be appended          |
| INPUT  | Specifies a sequential file which is to be read from         |
| OUTPUT | Specifies a sequential file which is to be written to        |
| RANDOM | Specifies a random-access file to be read from or written to |

*mode\_string* is a string expression of one character which is the first letter of the *mode* of the file (A, I, O or R), in upper- or lower-case.

*channel\_num* may be any integer value from 1 to 255 inclusive.

*record\_size* specifies the length of each record in bytes in a random-access file, or the internal buffer size for other types of file. The default value for *record\_size* is 128 bytes.

OPEN associates a file or device with *channel\_num*. This number is used in all read or write operations to access the file or device.

The counterpart to OPEN is CLOSE. It is a good idea to CLOSE a file whenever possible and reOPEN it later if necessary. This is a safety measure; if the system should go crazy for some reason, any data which is still in a buffer will be lost and the file may very well later contain garbage.

Normally *file\_spec* refers to a disk file, but there are certain device names that may also be used. These names are:

|      |  |
|------|--|
| CON: | screen and keyboard  |
| PRT: | parallel printer port                                      |
| AUX: | RS232 port   |
| LST: | current printer port as chosen with <i>Install Printer</i> |
| MID: | MIDI port  |
| KBD: | Intelligent keyboard                                       |

Note that channels opened with these names will only respond to simple input/output, that is PRINT#, WRITE# INPUT# and INPUT\$. Random access operations and other operations (such as LOF;) that do not make sense on non-disk devices, will not work. Care should be taken when opening the CON: device as if in GEM mode the window contents will be lost and if in TOS mode output from PRINT statements will become mixed up with output from PRINT#.

If you use OPEN on a channel that is already open the old file will be closed before the new one is opened. If the file was opened for random access any associated FIELDed variables will be cleared

- **See**

CLOSE, Fopen & FClose GEMDOS calls.

- **Example**

```
OPEN "RECORDS.RAN" FOR RANDOM AS #1 LEN=32
'This line opens the random-access file RECORDS.RAN for
'reading & writing, associates it with #1 and specifies
'the record length to be 32 bytes.
'the next line does the same using the alternate syntax
'except using #2
OPEN "R",#2,"RECORDS.RAN",32
```

## OPTION BASE statement

- **Syntax**

OPTION BASE {0|1}

- **Effect**

This statement defines the lowest subscript value of arrays.

- **Comments**

OPTION BASE sets the subscript value of the first element in an array. For instance, an OPTION BASE 1 before DIM Array%(42) causes 42 elements (1-42) to be allocated, not the default 43 (0-42).

Using OPTION BASE 1 saves hardly any space unless you are using three or higher dimension arrays.

Note that if the Array Checks option is off when a program is compiled, OPTION BASE statements are ignored and OPTION BASE 0 is assumed.

- **See**

DIM.

- **Example**

```
OPTION BASE 0      'this is default
DIM Another_array%(177)  '178 elements will be
                        'allocated, 0 to 177
```

## OUT statement

### • Syntax

OUT *BIOS\_handle*, *integer\_expression*

### • Effect

This statement sends a byte to the BIOS output routines.

### • Comments

*BIOS\_handle* may have the following values:

|   |                                 |
|---|---------------------------------|
| 0 | Parallel port                   |
| 1 | Serial port                     |
| 2 | Console                         |
| 3 | MIDI                            |
| 4 | Intelligent keyboard controller |
| 5 | Direct screen output            |

The range of the byte sent to the port is 0 to 255.

If you send bytes directly to the screen or printer using OUT you can confuse the BASICs idea of where the cursor is, for example.

Do not attempt to send bytes to the keyboard controller unless you are very sure what you are doing.

### • See

INP, BConout BIOS routine.

### • Example

```
PRINT "This action is ";
OUT 2,27: OUT 2,"p"% 'reverse video
PRINT "VERY DANGEROUS";
OUT 2,27: OUT 2,"q"% 'normal video
```

## PALETTE statement

### • Syntax

PALETTE [*colour\_number*,*physical\_colour*]

or

PALETTE USING *array\_element*

### • Effect

This statement allows you to change the physical appearance of colours on the screen.

### • Comments

Each colour on the ST can have its actual appearance changed on screen by changing the colour palette. If no parameters are given all colours are restored to their normal defaults, as shown in the table under the COLOR command. *colour\_number* should be an integer from 0 to 15, that corresponds to the colour you wish to change. The *physical\_colour* is the value to be placed into the register, in the form of three hex digits in the range 0 to 7, one for red, one for green, and one for blue.

To use the extended palette range of the TT and STE with this statement, the hex digits can also include 8 to F but in order to maintain compatibility they are mapped in the following way:

|                  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|------------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| STE              | 0 | 8 | 1 | 9 | 2 | A | 3 | B | 4 | C | 5  | D  | 6  | E  | 7  | F  |
| Intensity of gun | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| ST               | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |   |    |    |    |    |    |    |

The PALETTE USING command takes an array as a parameter and uses all the elements in the array and loads them into each colour.

We recommend using the GEMVDI *v\_color* routine as this lets you set the palette in a machine independent way without the complexity of the mapping above. You can also easily use the 256 colours of TT Low Resolution with this call, whereas to use the hardware directly you need to use the Esetpalette XBIOS routine.

### • See

*vs\_color* VDI routine, *setcolor*, *setpalette*, *esetcolor*, *esetbank*, *esetpalette* BIOS routines

### • Example

```
PALETTE          'reset to default
PALETTE USING a%(0) 'set them up
PALETTE 0,&h701    'make white into red with
                  'a little blue

INPUT "Invert screen";a$    'monochrome only
IF UCASE$(a$)="Y" THEN
    PALETTE 0,0    'make white=black
ELSE
    PALETTE 0,&h777 'make white=white
END IF
```

## PCIRCLE statement

### • Syntax

PCIRCLE *x\_centre*,*y\_centre*,*radius*[,*start\_angle*,*end\_angle*]

### • Effect

Draws a solid circle or arc in the current window.

### • Comments

If the optional angle parameters are omitted, the command draws a filled circle with an outline and fills it with the current fill colour, index and style attributes set with **COLOR**. The angle parameters are expressed in degrees times 10, and are measured anti-clockwise starting on the right side. Hollow circles and arcs can be drawn using the **CIRCLE** command. All the parameters are integers.

The *x\_centre* and *y\_centre* are relative to the current window rather than the screen. **PCIRCLE** ensures that the x radius and y radius are the same in pixels. As a result the graphic that is drawn will not appear circular in some resolutions (e.g. ST Medium) where the pixels are not square. To draw circles that look like circles but which will have different y-radii use the GEMVDI *v\_circle* command and then fill that circle.

### • See

PELLIPSE, CIRCLE, BAR, *v\_circle*, *v\_ellipse*, *v\_ellpie* (VDI routines).

### • Example

```
COLOR 1,1,1,6,2
PCIRCLE 200,100,50
```

## PCOPY statement

---

- **Syntax**

PCOPY

- **Effect**

Dump the current screen to a printer.

- **Comments**

This has the same effect as pressing Alt-Help, and may be interrupted by pressing the same key combination.

- **See**

LPRINT.

- **Example**

```
IF a$="C" THEN
  MOUSE -1      'hide mouse
  PCOPY          'and dump screen
END IF
```

## PEEK function

---

- **Syntax**

PEEK(address)

- **Effect**

This function returns a byte, word or long of the contents at location *address*.

- **Comments**

In general we recommend the use of the PEEKB, PEEKW, PEEKL, SPEEKB, SPEEKW or SPEEKL functions and their POKE counterparts rather than the use of DEF SEG, PEEK and POKE which are provided for ST BASIC compatibility. The new style statements are faster and easier to understand.

The length of the returned value (byte, word or long) depends on the DEF SEG statement.

If DEF SEG > 0 then 1 byte is PEEKed and the value of the *address* is added to the DEF SEG value.

If DEF SEG = 0 and the *address* given to PEEK is of type single or integer then 2 bytes are PEEKed from address.

If DEF SEG = 0 and the *address* passed is of type long integer or double, then 4 bytes are PEEKed.

This differs from the PEEKB, PEEKW and PEEKL statements in that it can read protected memory areas and words from odd addresses, but is much slower as a result.

- **See**

POKE, PEEKB, PEEKW, PEEKL, SPEEKB, SPEEKW, SPEEKL.

- **Example**

```
DEF SEG 0
PRINT PEEK(&H4A6)  'This returns the number of floppy
                   'disk drives connected
```



## PEEKB, PEEKL, PEEKW functions

### • Syntax

PEEKB(*address*)  
PEEKL(*even\_address*)  
PEEKW(*even\_address*)

### • Effect

These functions return the contents of the memory specified.

### • Comments

PEEKB returns one byte, PEEKW returns a word (16-bit) and PEEKL returns a long (32-bit). When using PEEKL or PEEKW, the address specified must be even, otherwise an address error (3 bombs) will occur unless you are running on a machine with at least a 68020 processor.

These functions are only able to read memory within user memory. Any read which requires supervisor mode or of non-existent memory will result in a bus error (2 bombs).

These functions are supplied as fast ways of reading user memory. For this reason, no checks are made as to the even-ness of the addresses for PEEKL and PEEKW; the same reason applies to no checks for reads of supervisor memory. To read from supervisor memory use the SPEEKB, SPEEKW and SPEEKL functions.

PEEKB and PEEKW return integers; PEEKL returns a long integer.

### • See

SPEEKB, SPEEKW, SPEEKL, POKEB, POKEW, POKEL.

### • Example

```
Seven%=7  
PRINT PEEKW(VARPTR(Seven%))
```

Result:

7

## PELLIPSE statement

### • Syntax

PELLIPSE *x\_centre,y\_centre,x\_radius,y\_radius*  
[,*start\_angle,end\_angle*]

### • Effect

Draws a solid ellipse or elliptical arc in the current window.

### • Comments

If the optional angle parameters are omitted, the command draws a solid ellipse in the current fill colour, index and style. The angle parameters are expressed in degrees times 10, and are measured anti-clockwise starting on the right side. Hollow ellipses and arcs can be drawn using the ELLIPSE command. All the parameters are integers.

The *x\_centre* and *y\_centre* are relative to the current window rather than the screen. unlike the corresponding *v\_ellpie* VDI command.

### • See

PCIRCLE, ELLIPSE, *v\_ellpie* GEMVDI command.

### • Example

```
COLOR 1,1,1,19,2  
PELLIPSE 200,100,100,30  
COLOR 1,1,1,4,2  
PELLIPSE 200,100,80,25,0,1800
```

## POINT function

### • Syntax

POINT(*x\_pixel*,*y\_pixel*)

### • Effect

This function returns the colour of a particular pixel.

### • Comments

*x\_pixel* and *y\_pixel* specify the coordinates within the current output window of the pixel to be read. The result corresponds to the table shown under the description of COLOR.

If the required pixel is not within the window, a result of -1 is returned.

### • See

PSET, PRESET, LINEF.

### • Example

```
COLOR 1,1,1,6,2
PRINT "B"
FOR x=0 to 8
FOR y=0 to 16
IF POINT(x,y) THEN BAR 10*(x+1),10*(y+2),10,10
NEXT y,x
```

## POKE statement

### • Syntax

POKE *address*,*data*

### • Effect

This statement writes *data* directly to *address* in memory.

### • Comments

In general we recommend the use of the POKEB, POKEW, POKEL, SPOKEB, SPOKEW or SPOKEL functions and their PEEK counterparts rather than the use of DEF SEG, PEEK and POKE which are provided for ST BASIC compatibility. The new style statements are faster and easier to understand.

The size of *data* that is written is dependent on the last DEF SEG statement. DEF SEG is zero by default.

If DEF SEG > 0 then 1 byte is POKed and the value of the *address* is used as an offset of the value of DEF SEG.

If DEF SEG = 0 and the *address* given to POKE is of type single or integer then 2 bytes POKed.

If DEF SEG = 0 and the *address* passed is of type long integer or double, then 4 bytes are POKed.

POKE is a good deal slower than POKEB, POKEW and POKEL but is able to write to protected or non word-aligned locations. Note that the value POKed does not depend on the *data* but on the *address*.

### • See

PEEK, POKEB, POKEW, POKEL, SPOKEB, SPOKEW, SPOKEL.

### • Example

```
DEF SEG 0      'POKE a word
POKE &H484,14  'this turns off the keypress click
```

## POKEB, POKEL, POKEW statements

---

- **Syntax**

POKEB *address, byte\_value*  
POKEL *even\_address, long\_value*  
POKEW *even\_address, word\_value*

- **Effect**

These statements write data directly into memory.

- **Comments**

These statements are very much faster than POKE. They can not POKE into supervisor memory, nor do they check for even addresses for POKEL and POKEW.

POKEtype into supervisor memory will cause a bus error (2 bombs). To poke into such memory use the SPOKEtype commands.

POKEL or POKEW to an odd address will result in an address error (3 bombs).

- **See**

POKE, POKEB, POKEW, POKEL, SPEEKB, SPEEKW, SPEEKL.

- **Example**

```
Ten%=10
POKEW VARPTR(Ten%),7
PRINT Ten%
```

Result:

7

## POS function

---

- **Syntax**

POS(*x*)

- **Effect**

This function returns the column number of the current cursor position.

- **Comments**

*x* is a dummy argument and serves no purpose. The value returned is the column (horizontal position) of the cursor, the leftmost position being 1. *x* should be numeric.

- **See**

LPOES, LOCATE, PRINT.

- **Example**

```
x%=POS(0)
PRINT "The cursor was in column";x%
```

## PRESET statement

### • Syntax

PRESET [STEP] (*x\_pos*,*y\_pos*) [,*colour*]

### • Effect

This statement resets or sets a pixel to a given colour in the current window.

### • Comments

PRESET is similar to PSET, except that if the *colour* parameter is omitted the background (or fill) colour is used. The parameters *x\_pos* and *y\_pos* specify the pixel set, relative to the top left of the current window. If the optional STEP keyword is specified then the co-ordinates are taken relative to the last PSET or PRESET co-ordinates.

### • See

PSET, POINT, LINEF, v\_pline GEMVDI routine.

### • Example

```
CONST ypos=70
DEFINT i
FOR i=10 TO 350
    IF i<300 THEN PSET (i,ypos)
    IF i>50 THEN PRESET (i-50,ypos)
NEXT i
```

## PRINT statement

### • Syntax

PRINT [*expression\_1*][{;| |,}*expression\_2*][;]...

### • Effect

This statement prints the data defined in the list of expressions on the screen.

### • Comments

The list of expressions may contain a series of string and/or numeric expressions. These expressions must be separated with semi-colons, commas or spaces. If the list of expressions does not end in a semi-colon or a comma, a CR-LF will be output.

A PRINT by itself will cause a CR-LF.

A semi-colon separating expressions will cause them to be printed without a space between them. A space between parameters has the same effect as a semi-colon. We recommend using a semi-colon. A comma will cause the expression after the comma to be printed at the next tab stop.

SPC and TAB may be used in PRINT statements, but not elsewhere.

Numbers are printed with either a leading space (if positive) or minus sign, and always with a trailing space.

A question-mark may be substituted for PRINT if absolutely necessary; this is a feature of some older BASIC interpreters. To enhance compatibility with other BASICs, we have included this option as well.

PRINT in TOS mode gives 80 columns by 25 rows in ST High and Medium resolutions.

PRINT in GEM mode gives rather fewer characters, firstly because of the space taken by the menu bar and window title and scroll bars. You can 'reclaim' this space by using WINDOW FULLW. The second reason for fewer characters on the screen is that, for compatibility with older BASICs, HiSoft BASIC leaves a margin of one pixel at the top and bottom of each window and a gap of one pixel between each line.

You can remove these using the SYSTAB function as follows:

```
' decrease character height
POKEW SYSTAB+38,PEEKW(SYSTAB+38)-1
POKEL SYSTAB+72,0 ' make both margins 0
```

### • See

OUT, LPRINT, PRINT #, PRINT USING, WRITE.

### • Example

```
PRINT "The range is";1;"to";100,"(inclusive)"
```

Result:

The range is 1 to 100 (inclusive)

## PRINT#, PRINT# USING statements

---

### • Syntax

```
PRINT #channel_num, [USING  
format_string] expression_list [;]
```

### • Effect

These statements write formatted data to a file or device.

### • Comments

*channel\_num* is the number specified when the file was opened.

*format\_string* is an optional string of formatting characters. For an explanation of these, please refer to the PRINT USING entry.

*expression\_list* is the list of string and/or numeric expressions to be written.

PRINT# writes the data exactly the way PRINT would to the screen. For this reason, great care should be taken to format the data written so that it is recognisable for a later INPUT#.

### • See

PUT, WRITE, INPUT#, FORMATx\$.

### • Example

```
FOR I%=1 TO 5  
PRINT #1,I%;" , ";  
NEXT I%
```

Result: (in the target file)

1 , 2 , 3 , 4 , 5 ,

# PRINT USING statement

## • Syntax

PRINT USING *format\_string*; *expression\_list* [{,|;}]

## • Effect

This statement prints *expression\_list* according to the format specified by *format\_string*.

## • Comments

*expression\_list* contains the expressions (numeric or string) that are to be printed; they must be separated by semi-colons.

*format\_string* is a string of formatting characters which determine the field and format of the expressions to be printed.

There are three format characters available for formatting strings:

- ! This specifies that only the first character of the string is to be printed.
- \ \ These specify that  $2 + n$  characters of the string are to be printed.  $n$  is the number of spaces between the backslashes. If the string is longer than  $2 + n$ , the extra characters are not printed. If  $2 + n$  is larger than the number of characters in the string, the string will be printed left-justified with spaces padding the right.
- & This specifies a variable length field; if the ampersand (&) is the *format\_string*, the string is printed without modification.

There are several format characters available for formatting numeric output:

- # The hash represents each digit to be printed. These positions are always filled; if a number has less digits than have been specified, the number will be printed right justified with spaces padding the left.
- .
- A decimal point is printed. If *format\_string* specifies that a digit precedes the decimal, a digit will always be printed, even if is zero.
- + This causes the sign of the number (+/-) to be printed at the beginning or end of the string, depending on whether the + is at the beginning or end of *format\_string*.

- When placed at the end of *format\_string*, this causes a minus sign to be printed if the number is negative. If the number is positive, this has no effect.
- \*\* This causes leading spaces of a number to be filled with asterisks. A double asterisk also represents two more digit positions.
- \$\$ The double dollar-sign causes a dollar-sign to be printed to the immediate left of the number; two more digit positions, one of which is for the \$, are specified.
- \*\*\$ This results in a combination of \*\* and \$\$; leading spaces are asterisks and a \$ is printed before the number. Three more digit positions are specified, one of which is taken up by the dollar-sign. If a number is negative, a minus sign immediately precedes the \$.
- ,
- If specified before a decimal point, a comma is printed every three digits to the left of the decimal. If , is specified at the end of *format\_string*, it is printed as part of the number. The comma specifies one additional digit position and has no effect when used together with ^^^^.
- ^^^^ The four carets specify exponential format. Space is reserved for E+xx to be printed. the decimal point may be specified to be anywhere in the number. The significant digits of the number are left justified. Unless a leading/trailing + or a trailing - are specified, one digit position to the left of the decimal point will be used to print the number's sign. Note that double-precision numbers can have five digit exponents, so five carets should be used.
- \_ An underscore before a character in the *format\_string* causes the character to be printed as a literal; an underscore will be printed by specifying two consecutive underscores in the *format\_string*.

A percent sign is printed, if the number to be printed exceeds the field in the *format\_string*. If a rounded number is larger than the field, a percent sign will be printed in front of the number.

The currency symbol that is printed when using the \$ formats above. To do this you should change the byte at SYSTAB+66. For example,

```
POKEB SYSTAB+66,ASC("£")
```

will change it to the U.K. pound sign.

- **See**

PRINT, WRITE, FORMATx\$.

- **Example**

```
PRINT USING "!"; "FRED"
PRINT USING "\ "; "JOHN"
PRINT USING "&"; "Today is"
PRINT USING "###.#"; 25.4
PRINT USING "#.#"; 2.54
PRINT USING "###.## "; 12.34, 5.678, 1.2
PRINT USING "+#.## "; -1.1, 2.2
PRINT USING "#.#- "; -3.3, 4.4
PRINT USING "**#.## "; 56.78, -91.2, -3
PRINT USING "$$###.#"; 123.45
PRINT USING "***$###.##"; 2.54
PRINT USING "#####.##"; 4567.89
PRINT USING "+#####^"; -0.4444
PRINT USING " £###.##"; 4.99
PRINT USING "#.#"; 33.25
PRINT USING ".##"; .999
```

Results:

```
F
JOH
Today is
 25.4
 2.5
12.34  5.68  1.20
-1.1 +2.2
3.3- 4.4
*56.8 -91.2 *-3.0
$123.5
***$2.54
4,567.89
-4444E-04
£ 4.99
%33.3
.99
```

## **PSET statement**

- **Syntax**

PSET [STEP] (*x\_pos*, *y\_pos*) [, *colour*]

- **Effect**

This statement plots a pixel of a given colour in the current window.

- **Comments**

The *colour* parameter specifies the colour of the pixel to be plotted, and if omitted defaults to the current line colour. The parameters *x\_pos* and *y\_pos* specify the pixel to be plotted, relative to the top left of the current window. If the optional STEP keyword is specified then the co-ordinates are taken relative to the last PSET or PRESET co-ordinates.

- **See**

PRESET, POINT, BAR, v\_pline GEMVDI routine.

- **Example**

```
pi=3.1415926
FOR i=0 to 4*pi STEP 0.05
    PSET (i*50, 100+100*SIN(i))
NEXT i
```



## PUT file I/O statement

### • Syntax

PUT [#]channel\_number[,record\_number]

### • Effect

This statement writes a record from the random-access buffer to the designated random-access file.

### • Comments

channel\_number is the number specified when the file was opened.

If record\_number is not specified, the record written will be the next record.

### • See

GET, OPEN, CLOSE, *User Manual : Concepts* chapter.

### • Example

```
LSET a$=address$(10)
PUT #1,10
LSET a$=address$(11)
PUT #1          'will be record 11 by default
```

## PUT graphics statement

### • Syntax

PUT (x,y),array\_name[(arrayindex...)][,verb]

### • Effect

This statement copies (or *blits*) onto the screen a rectangular image saved with GET.

### • Comments

The top left of the rectangle relative to the screen is defined by *x* and *y*. If the array was not filled by a GET then unpredictable and potentially disastrous events may occur.

The optional *verb* parameter allows the mode of the copy to be specified. This can be in the form of various reserved words, or an integer. Allowed verbs are:

|              |  |
|--------------|--|
| PSET         | Copies the image directly onto the screen, obliterating any previous contents.                   |
| PRESET       | Similar to PSET but a negative image results.  |
| AND, OR, XOR | Resulting pixels are a result of the given operation performed on source and destination pixels. |

The default verb is XOR. The verb may also be described in terms of an integer expression, the effect of which is described in *Chapter 2 GEM VDI: Raster Functions*.

The height and width that are used are stored when the image is saved with GET.

The GET command acts on the *current* screen; to change the current screen (e.g. for animation) you should update the long word at SYSTAB+50.

Multi-dimensional arrays may be used in this statement, making it easy to switch between different images.

### • See

GET, vro\_cpyfm GEMVDI call.

### • Example

```
DIM scr%(4347-1) ' -1 as array starts at 0
GET (10,20)-(100,200),scr%
FOR i%=0 TO 200 STEP 20
    PUT (i%,i%),scr%,PSET
NEXT i%
```

## RANDOMIZE statement

- **Syntax**

RANDOMIZE [*expression*]

- **Effect**

This statement provides the random number generator with a new seed.

- **Comments**

If no *expression* is supplied, RANDOMIZE will prompt the user for a seed. *expression* may be any integer expression. If TIMER is given as *expression*, the TIMER function will provide the seed.

The RND function will produce the same sequence of random numbers unless re-seeded.

Note that the same seed will give exactly the same random number sequence each time.

- **See**

RND.

- **Example**

```
Seed%=42
RANDOMIZE Seed%
Rand!=RND
```

## READ statement

- **Syntax**

READ *variable\_list*

- **Effect**

This statement assigns values from DATA statements to variables.

- **Comments**

READ cannot be used without a DATA statement. The variable type in a READ statement will be converted to the variable type in the DATA statement from which the value is read.

The number of elements in a DATA statement should not exceed the number of variables in the READ statement. If there are more elements in a DATA statement than are READ, the extra elements are ignored.

To re-READ the elements of DATA statements, the RESTORE statement is used.

- **See**

DATA, INPUT\$, INPUT.

- **Example**

```
DIM Arr&(5)
FOR I%=1 TO 5
    READ Arr&(I%)
NEXT I%
DATA &H42424242,&HFC0000,1200,-45002
```

## REDIM statement

### • Syntax

```
REDIM [PRESERVE|APPEND]  
      array(subscripts)[,array(subscripts)]...
```

### • Effect

This statement changes the size of a dynamic array.

### • Comments

*subscripts* may be integer or long integer expressions. REDIM can even be used to change the number of dimensions in an array. Any data that is in an array when it is REDIMmed is lost.

Arrays that are to be REDIMed must be dynamic. That is they must be originally DIMed with an expression (not just a constant) or REM \$DYNAMIC must be in effect when they are originally DIMmed.

REDIM PRESERVE is used to add space to a *one-dimensional* array without destroying the information contained in the array. If the subscript of a REDIM PRESERVE is smaller than the original array, the array will be truncated; this also happens without loss of data in the remaining elements.

REDIM APPEND is equivalent to REDIM PRESERVE for compatibility with HiSoft BASIC version 1. REDIM PRESERVE is to be preferred since this is used by MicroSoft BASIC 7.1 on the PC.

### • See

DIM, ERASE, *User Manual* : *Concepts* chapter.

### • Example

```
DIM Fred%(256)  
FOR I%=1 TO 256  
    Fred%(I%)=I%  
NEXT I%  
REDIM PRESERVE Fred%(260)  
PRINT Fred%(42)
```

Result:

42

## REM statement

### • Syntax

```
REM remarks...
```

### • Effect

This statements allows remarks to be added within the source, and also to specify meta-commands.

### • Comments

Comments can also begin with ' (i.e. apostrophe). See the *Concepts* chapter in the *User Manual* for details.

If a dollar sign (\$) follows the REM it is assumed to be a meta-command (see *Appendix B* and *Chapter 3* of the *User Manual* for further details), otherwise the rest of the line is ignored.

### • See

*Appendix B*.

### Example

```
' $OPTION A+  
REM The preceding line forces array checks on
```

## REPEAT...END REPEAT statement

### • Syntax

```
REPEAT name
.  
.  
[EXIT name]  
.  
.  
END REPEAT name
```

### • Effect

The statements within the REPEAT loop are executed until an EXIT statement for the loop is executed.

### • Comments

The *name* is a valid variable name that is not currently being used in the program. You can re-use the same name for a number of different REPEAT loops but do not use the name of a sub-program, variable or function.

This loop is the most general looping construct in HiSoft BASIC because each loop is named and you can exit out of an outer loop from within an inner one (see the example below). However when this facility is not required DO...LOOPs are normally clearer, require less typing and are compatible with Microsoft BASIC as well.

Note that REPEAT is *not* specified in the EXIT statement.

### • See

DO...LOOP, WHILE...WEND, FOR...NEXT.

### • Example

```
i=0: sum=0
REPEAT one
  i=i+1: j=0
  REPEAT two
    j=j+1
    sum=sum+a(i,j)
    IF sum>10000 THEN
      PRINT "Total too big": EXIT one
    END IF
    IF j=n THEN EXIT two
  END REPEAT two
  IF i=n THEN EXIT one
END REPEAT one
' This example totals the elements in the two-dimensional
' array a() stopping immediately when the sum becomes
' greater than 10000.
```

## RESET statement

---

- **Syntax**

RESET

- **Effect**

This statement closes all open disk files.

- **Comments**

All disk write buffers are written to disk and the files are then closed. Since all files are closed automatically on exit you normally do not need to use this command. It is provided for compatibility with other BASICs.

- **Example**

IF a\$="QUIT" THEN RESET : SYSTEM ' the RESET is redundant

## RESTORE statement

---

- **Syntax**

RESTORE [{*line\_number*|*label*}]

- **Effect**

This statements enables a READ statement to access a DATA statement that has been read previously.

- **Comments**

*line\_number* or *label* specify the DATA statement that is to be READ again. If no *line\_number* or *label* is specified, the next READ statement will read the first DATA statement to appear in the program.

- **See**

READ, DATA, INPUT.

- **Example**

```
READ John&
PRINT John&
John&=0
PRINT John&
RESTORE J_data
READ John&
PRINT John&
.
.
J_data:
DATA 3448325377,4020987683
```

Result:

```
3448325377
0
3448325377
```

## RESUME statement

### • Syntax

RESUME {*line\_number*|*line\_label*}

### • Effect

This statement resumes program execution from within an error handling routine at the specified program line.

### • Comments

This command is for use from error handling routines set up with ON ERROR GOTO to resume execution of the program at the given position. If the statement is used when not in an error-handler a fatal error will occur.

It is the programmers responsibility that the RESUME is to the same program level as when the error occurred - that is if it was in a sub-program, function or subroutine the RESUME *must* go there for correct program execution. Failure to adhere to this will result in random forms of program crashes.

### • See

ON ERROR, ERL, ERROR.

### • Example

```
a$="TEST.DAT"
open_retry:
  ON ERROR GOTO no_open
  OPEN a$ FOR INPUT AS #1
  ON ERROR GOTO 0 'so that errors are reported
  read_data 1
  .....
no_open:
  PRINT "Cannot find file ";a$
  INPUT "Filename:";a$
  RESUME open_retry
```

## RETURN statement

### • Syntax

RETURN [{*line\_number*|*label*}]

### • Effect

This statement returns program execution from a subroutine.

### • Comments

Program execution is returned to the line after the GOSUB statement that called the subroutine. If *line\_number* or *label* are specified, execution continues on the specified line.

Note that this statement does *not* exit out of sub-programs, use EXIT SUB for this.

### • See

GOSUB.SUB, END, EXIT.

### • Example

```
GOSUB Routine      'Branch to subroutine
.                  'execution continues here
.                  'after the RETURN
Routine:
.
.
RETURN             'return from subroutine
```

## **RIGHT\$ function**

---

- **Syntax**

RIGHT\$(*string\_expression*,*n*)

- **Effect**

This function returns a string starting at the *n*th character from the right.

- **Comments**

If *n* is 0, a null string is returned. If *n* is larger than *string\_expression*, *string\_expression* is returned entirely.

- **See**

LEFT\$, MID\$, LEN.

- **Example**

```
d$=DATE$  
PRINT "Year: ";RIGHT$(d$,4)
```

## **RMDIR statement**

---

- **Syntax**

RMDIR *pathname*

- **Effect**

This statement removes an empty subdirectory.

- **Comments**

The subdirectory which is to be deleted must not contain any files. *pathname* must conform to the GEMDOS conventions.

- **See**

KILL, MKDIRE, CHDIR.

- **Example**

```
KILL "\GARBAGE\*.*" 'remove all files  
RMDIR "\GARBAGE"    'then the directory itself
```



## RND function

---

- **Syntax**

RND[ (*n*) ]

- **Effect**

This function returns a pseudo-random single-precision number between 0 and 1.

- **Comments**

If  $n = 0$ , the last number generated is returned. If  $n < 0$ ,  $n$  is used to re-seed the sequence of numbers. If  $n > 0$  or is omitted, the next number in the sequence is generated.

The algorithm used is based on that of B.A. Wichman and I.D. Hill (NPL Report DITC 6/82). See also the RANDOMIZE statement.

- **See**

RANDOMIZE.

- **Example**

```
Number!=RND(-1) 'the number returned will have  
                'used -1 as the seed
```

## RSET statement

---

- **Syntax**

RSET *string\_variable* = *string\_expression*

- **Effect**

This statement moves data into a random-access file buffer. It can also be used to right-justify the value of the string in *string\_variable*.

- **Comments**

RSET is the counterpart of LSET. If *string\_expression* uses less bytes than *string\_variable*, the string is right-justified with spaces used as padding.

- **See**

LSET, GET, FIELD.

- **Example**

```
z$=SPACE$(80)  
RSET z$="On the Right of the Screen"  
PRINT z$
```

## RTRIM\$ function

---

- **Syntax**

RTRIM\$(*string\_expression*)

- **Effect**

This function returns its string parameter with any trailing spaces removed.

- **Comments**

RTRIM\$ trims spaces from the right. This can be used to remove spaces at the end of user input.

- **See**

LTRIM\$, RIGHT\$.

- **Examples**

```
commandline$=RTRIM$(LTRIM$(COMMAND$))  
' commandline$ won't have spaces at the front or the end
```

```
PRINT RTRIM$("  HiSoft  ");RTRIM$("BASIC")
```

Result:

```
HiSoftBASIC
```

## RUN statement

---

- **Syntax**

RUN [{*line\_number*|*file\_spec*}]

- **Effect**

This statement restarts the current program or the program specified by *file\_spec*.

- **Comments**

If *line\_number* is specified, the program is restarted on the line specified. Alphanumeric labels are not allowed.

*file\_spec* can be any valid string expression, and can be any type of executable program. In some BASICs filenames do not need to be in quotes, but they are needed in HiSoft BASIC. This facility should not be used when compiling to memory.

RUN will close all open files and clear current user memory before executing.

- **See**

CHAIN, Execute HGT command, pexec GEMDOS call.

- **Example**

```
RUN "GEN.PRG" 'executes Gen from the current directory
```

## SADD function

---

### • Syntax

SADD(*string\_expression*)

### • Effect

This function returns the address of *string\_expression*.

### • Comments

Be very careful when using this function as strings can move about in memory without prior warning if a garbage collect should occur. Thus normally SADD present in the statement that is using the address rather than assigned to a temporary variable.

### • See

VARPTR.

### • Example

```
Astr$=" BASIC"
PRINT Astr$
POKEB SADD(Astr$),ASC("H")
PRINT Astr$
```

Result:

```
BASIC
HBASIC
```

## SCREEN statement

---

### • Syntax

SCREEN *mode*

### • Effect

This statement changes the screen mode.

### • Comments

This statement will only work with a colour monitor attached. The available modes are:

- 0 ST low resolution
- 1 ST medium resolution
- 2 ST high resolution (only available on TT colour monitors)
- 4 TT medium resolution
- 7 TT low resolution'

Naturally the TT resolutions require a TT. An **Illegal function call** in module SCREEN if you try to access a mode that is not available.

A *mode* of 1 selects ST medium resolution, and a *mode* of 0 selects ST low resolution.

Note that this should only be used from TOS- or TTP-type programs - the GEM AES and VDI cannot cope with a screen mode change from within a program. To change screen modes in a way GEM recognises you have to use the *Set Preferences* option from the Desktop.

Having changed resolutions you may need to change the colour palettes, using PALETTE.

### • Example

```
IF getrez%=0 THEN
  SCREEN 1      'go into medium res
  PALETTE
  CLS
  WIDTH 80
END IF
```

# SELECT...END SELECT statement

## • Syntax

SELECT [CASE|ON] *variable*

[CASE|=] *case\_list*  
[EXIT SELECT]

[CASE|=] *case\_list*

[CASE|=] [ELSE|REMAINDER]

END SELECT

## • Effect

One series of statements is executed depending on the value of *variable*.

## • Comments

*variable* can be any non-subscripted variable, numeric or string. If you wish to select on a more complicated expression then this should be assigned to a temporary variable first.

The *case\_lists* consist of the following items separated by commas:

*expression*

or

*first\_expression* TO *second\_expression*

or

*relop expression*

or

IS *relop expression*

If the first is used then if the *variable* is equal to the *expression* then the following statements are executed until the next CASE or = statement.

If the second form is used, then if the *variable* is greater or equal to the *first\_expression* and less than or equal to the *second\_expression* then the following statements are executed.

In the third and fourth forms *relop* may be one of =,<>,>,>=,<=,< or == and the following statements are executed if the *variable* satisfies the condition.

If there is a CASE ELSE, CASE REMAINDER, =REMAINDER or =ELSE clause then if none of the CASEs match then the following statements will be executed. The CASE ELSE clause must be the last before the END SELECT.

The different alternatives are available for ease of converting programs written in other BASICs. The MicroSoft BASIC form is SELECT CASE...CASE...CASE ELSE... END SELECT.

## • See

IF...THEN...ELSE.

## • Examples

```
INPUT A,B
SELECT CASE A
CASE 1,IS <0
    PRINT "negative or 1"
CASE IS >B
    PRINT "greater than B"
CASE 13 TO 19
    PRINT "in the teens"
CASE ELSE
    PRINT "different"
END SELECT
INPUT a$
SELECT a$
="JOHN","FRED": PRINT "John or Fred"
=<"A",>"Z": PRINT "Names start with a capital letter"
="MARY": PRINT "Mary"
= REMAINDER
    PRINT "unknown name"
END SELECT
```

## SGN function

### • Syntax

SGN(*numeric\_expression*)

### • Effect

This statement returns the sign of *numeric\_expression*.

### • Comments

If *numeric\_expression* < 0, -1 is returned.

If *numeric\_expression* = 0, 0 is returned.

If *numeric\_expression* > 0, 1 is returned.

*numeric\_expression* may be of any numeric type.

### • See

ABS.

### • Example

```
Var_one% = -42
Var_two% = 0
Var_thr% = 42
PRINT SGN(Var_one%)
PRINT SGN(Var_two%)
PRINT SGN(Var_thr%)
```

Result:

```
-1
0
1
```

## SHARED statement

### • Syntax

SHARED *variable\_list*

### • Effect

This statement enables a subprogram to access variables from the main program without them having been passed as parameters.

### • Comments

The *variable\_list* may contain variable names and array names, which should be terminated with ( ).

### • See

DIM, STATIC, LOCAL, COMMON SHARED.

### • Example

```
MAIN:
Fred%=127
John%=76
CALL Routine(John%,10)
.
.
SUB Routine (A%,B%)
SHARED Fred%
    Print Fred%,A%,B%
END SUB
```

Result:

```
127    76    10
```

## SIN function

### • Syntax

SIN(*numeric\_expression*)

### • Effect

This function returns the sine of the *numeric\_expression* which must be in radians.

### • Comments

This function is calculated with single-precision unless the *numeric\_expression* is in double precision, thus forcing a double-precision calculation.

### • See

COS, TAN, ATN.

### • Example

```
DEF cosecant(a)=1/SIN(a)
```

## SOUND statement

### • Syntax

SOUND *voice*,*volume* [,*note*][,*octave*][,*duration*]

### • Effect

This statement allows control over the 3 sound channels.

### • Comments

*voice* is the sound channel, from 1 to 3.

*volume* controls the output for that channel, from 0 (off) to 15 (loudest)

*note* and *octave* control the pitch of the note. The octave ranges from 1 to 8 and the note from 1 to 12.

*duration* is the time the note will continue before the beginning of the next sound, in 50ths of a second.

The last sound statement for each channel should turn off the sound, by specifying all other parameters to be zero.

Note that pressing a key will normally kill any sound being made, unless the keyclick feature is disabled with the *Control Panel* or XBIOS call.

| C | C# | D | D# | E | F | F# | G | A  | A# | B  |
|---|----|---|----|---|---|----|---|----|----|----|
| 1 | 2  | 3 | 4  | 5 | 6 | 7  | 8 | 10 | 11 | 12 |

Note Table

### • Example

```
PRINT "Close encounters..."
SOUND 1,15,10,3,20
SOUND 1,15,12,3,20
SOUND 1,15,8,3,20
SOUND 1,15,8,2,24
SOUND 1,15,3,3,40
SOUND 1,0
```

## SPACE\$ function

---

- **Syntax**

SPACE\$(*n*)

- **Effect**

This function returns a string of spaces (ASCII 32), the length of which is specified by *n*.

- **Comments**

*n* must be positive, and within the range of long integers. If merely want to print some spaces it is more efficient to use the SPC function.

- **See**

STRING\$.

- **Example**

```
DIM names$(100)
FOR i=0 TO 100
    name$(i)=SPACE$(30)
NEXT i
```

## SPC function

---

- **Syntax**

SPC(*n*)

- **Effect**

This function causes *n* spaces to be skipped in a PRINT statement.

- **Comments**

In contrast to SPACE\$, SPC may only be used with PRINT and LPRINT. *n* must be positive and within the long integer range. It is not necessary to specify a semi-colon after a SPC, the function will not produce any other characters except the *n* spaces.

In general it is best to use SPACE\$ in case you wish to convert the PRINT statement into another statement at a later date.

- **See**

PRINT, SPACE\$, STRING\$.

- **Example**

```
LOCATE 10,1
INPUT "Name: ",a$
LOCATE 10,1
PRINT SPC(5+LEN(a$))
LOCATE 10,1
INPUT "Address: ",b$
```

## **SPEEKB, SPEEKL, SPEEKW functions**

---

### **• Syntax**

SPEEKB(*address*)  
SPEEKL(*even\_address*)  
SPEEKW(*even\_address*)

### **• Effect**

These functions return the contents of the memory specified using the processor's supervisor mode.

### **• Comments**

SPEEKB returns one byte, SPEEKW returns a word (16-bit) and SPEEKL returns a long (32-bit). On the 68000, when using SPEEKL or SPEEKW, the address specified must be even, otherwise an address error (3 bombs) will occur.

These functions are able to access the hardware registers and low system memory, and thus care is required in their use.

These functions are slower than the corresponding user mode functions PEEKB, PEEKW and PEEKL because of the overhead of entering and leaving supervisor mode.

These functions are supplied a direct interface to the processor instructions. For this reason, no checks are made as to the even-ness of the addresses for SPEEKL and SPEEKW.

SPEEKB and SPEEKW return integers; SPEEKL returns a long integer.

### **• See**

PEEKB, PEEKW, PEEK, SPOKEB, SPOKEW, SPOKEL.

### **• Example**

```
initial200&=SPOKEL(&h4BA)
' initial200& will be current value of the 200Hz timer
```

## **SPOKEB, SPOKEL, SPOKEW statements**

---

### **• Syntax**

SPOKEB *address*, *byte\_value*  
SPOKEL *even\_address*, *long\_value*  
SPOKEW *even\_address*, *word\_value*

### **• Effect**

These statements write data directly into memory in supervisor mode..

### **• Comments**

These statements are slower than POKEB, POKEW and POKEL but they may be used to access supervisor memory and the hardware registers directly.

This is not something that we encourage since it is prone to error, may cause unexpected side effects and may not work as expected on future hardware. In general it is much better to use the operating system calls.

SPOKEL or SPOKEW to an odd address will result in an address error (3 bombs).

### **• See**

PEEKB, PEEKW, PEEK, SPOKEB, SPOKEW, SPOKEL.

### **• Example**

```
conterm=SPEEKB(&h484)
```

```
' clear bit 0 ie disable key click
SPOKEB &h484,conterm AND &hFE
```



## SQR function

---

- **Syntax**

SQR(*numeric\_expression*)

- **Effect**

this function returns the square root of the *numeric\_expression*.

- **Comments**

The *numeric\_expression* must be greater than or equal to 0. SQR is normally calculated using single precision maths; if the *numeric\_expression* is double-precision, the value returned will be calculated with double-precision.

- **Example**

```
PRINT SQR(42)
PRINT SQR(42#)
```

Result:

```
6.480741
6.48074069840786
```

## STATIC statement

---

- **Syntax**

STATIC *variable\_list*

- **Effect**

This statement declares variables as local to function definitions and sub-programs and preserves their values when the function or sub-program is left and re-entered.

- **Comments**

This statement can be used only within function definitions and sub-programs.

Variables in DEF FN function definitions usually are global. STATIC makes these variables local to the specific function.

We recommend that you use the STATIC statement even when this is the default as it means that you can use the Variable Checks (V) option.

- **See**

DIM, LOCAL, SHARED, COMMON SHARED.

- **Example**

```
test=10
PRINT FNeasy,test
DEF FNeasy
  STATIC test          'without this the global
  test=5               'test would be changed here
  FNeasy=1/test
END SUB
```

# STICK function

## • Syntax

STICK(*n*)

## • Effect

This function returns the x and y positions of the two joysticks.

## • Comments

*n* can be:

- |    |  |
|----|--|
| 0  | This returns the x position of joystick 0.     |
| 1  | This returns the y position of joystick 0.     |
| 2  | This returns the x position of joystick 1.     |
| 3  | This returns the y position of joystick 1.     |
| 4  | This returns the x position of STE joystick 1. |
| 5  | This returns the y position of STE joystick 1. |
| 6  | This returns the x position of STE joystick 2. |
| 7  | This returns the y position of STE joystick 2. |
| 8  | This returns the x position of STE joystick 3. |
| 9  | This returns the y position of STE joystick 3. |
| 10 | This returns the x position of STE joystick 4. |
| 11 | This returns the y position of STE joystick 4. |

Note: these functions only work after a STRIG ON command has been executed for the standard joysticks.

For the x positions the function returns:

-1: Right    0: Centre    1: Left

For the y positions the function returns:

-1: Down    0: Centre    1: Up

When the x position is requested the y position is also stored at the same time, so the program obtains simultaneous results.

Note that with a mouse connected to port 0 the results from this function are not particularly meaningful or useful.

The STE versions of these commands require that the STE hardware is present.

## • See

STRIG, MOUSE.

## • Example

```
PSET (200,100)
STRIG ON
DO
    XA% = STICK(2) 'the x position for joystick
                  '1 is returned and the y
                  'position stored
    YA% = STICK(3)
    PSET STEP (XA%,-YA%) 'plot relatively
    LOOP UNTIL STRIG(3) 'until Fire button pressed
STRIG OFF
```

## STOP statement

### • Syntax

STOP [*process\_returncode*]

### • Effect

This statement causes the program to end; all files are closed and control is returned to the operating system.

### • Comments

If no parameter is specified then the message Program terminated will appear at the bottom of the screen and it will pause until a key is pressed. To disable this specify a parameter of -1 or use the SYSTEM command.

If a parameter other than -1 is specified this is used as a GEMDOS process return code, useful for TOS programs that are to be run from CLIs, for example. By default the return code is 0 (no error), or the error code+200 if a run-time error occurs.

To return a negative return code use the SYSTEM statement.

### • See

SYSTEM, END.

### • Example

```
IF FEXISTS(COMMAND$) THEN
    process COMMAND$
    STOP -1      'will return immediately
ELSE
    PRINT "Cannot find ";COMMAND$
    STOP        'will wait for key
END IF
```

## STR\$ function

### • Syntax

STR\$(*numeric\_expression*)

### • Effect

This function returns the string representation of the value of *numeric\_expression*.

### • Comments

STR\$ is the counterpart to VAL.

If *numeric\_expression*  $\geq 0$ , the string returned contains a leading space, else it will start with a minus sign. *numeric\_expression* may be of any numeric type.

### • See

VAL, FORMATx\$.

### • Example

```
A%=4
B%=6
WRITE STR$(A%*B%)
```

Result:

" 24"

## STRIG function

### • Syntax

STRIG(*n*)

### • Effect

This function returns the status of the specified joystick fire button.

### • Comments

*n* may be:

- 0 This returns -1 if the button of joystick 0 was pressed since the last STRIG(0); 0 is returned if this is not the case.
- 1 -1 is returned if the button of joystick 0 is pressed, 0 if not.
- 2 This returns -1 if the button of joystick 1 was pressed since the last STRIG(2); 0 is returned if this is not the case.
- 3 -1 is returned if button of joystick 1 is currently pressed, 0 if not.
- 4 -1 is returned if button of STE joystick 1 is pressed, 0 if not.
- 5 -1 is returned if button of STE joystick 2 is pressed, 0 if not.
- 6 -1 is returned if button of STE joystick 3 is pressed, 0 if not.
- 7 -1 is returned if button of STE joystick 4 is pressed, 0 if not.

This function will only work after a STRIG ON statement has been executed for the standard joysticks.

The STE versions of these commands require that the STE hardware is present.

Note that the left mouse button is equivalent to joystick 0's fire button and the right button to joystick 1's.

### • See

STRIG statement, STICK.

### • Example

```
IF STRIG(1) = -1 THEN
  PRINT "Button A is being pressed"
ELSEIF STRIG(3) = -1 THEN
  PRINT "Button B is being pressed"
END IF
```

## STRIG statement

### • Syntax

STRIG ON

STRIG OFF

### • Effect

This function controls use of the joystick commands.

### • Comments

In order to read the joystick ports of the ST it is necessary to turn off mouse movement. The command STRIG ON does this, so that the STICK and STRIG functions will work. The command STRIG OFF will turn mouse movement back on.

Joystick functions will not work before a STRIG ON or after a STRIG OFF.

### • See

STICK, MOUSE.

### • Example

```
INPUT "Mouse or Joystick";choice$
SELECT CASE choice$
CASE "M"
  mouse_flag=1
CASE "J"
  mouse_flag=0
  MOUSE -1
  STRIG ON
END SELECT
```

## STRING\$ function

---

- **Syntax**

STRING\$(*m*,*n*)

STRING\$(*m*,*string\_expression*)

- **Effect**

This function returns a string composed of ASCII *n*, which is of *m* length.

- **Comments**

If *string\_expression* is specified instead of *n*, the first character of *string\_expression* is used.

- **See**

SPACE\$.

- **Example**

```
PRINT STRING$(40,127)
```

Result:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

## SUB...END SUB statement

---

- **Syntax**

```
SUB global_name[(parameter_list)] [STATIC]
```

```
:
```

```
:
```

```
[EXIT SUB]
```

```
:
```

```
:
```

```
END SUB
```

- **Effect**

This statement is used to define a sub-program.

- **Comments**

*global\_name* must be a name which is unique throughout the entire program. It is the name which refers to the sub-program.

*parameter\_list* is the list of parameters that are passed to a sub-program. These may be variable names, array names terminated with ( ), or literal values. If preceded by the keyword **VAL** then the parameter is always passed by value; this makes execution of the sub-program quicker.

**STATIC** is the optional attribute which signifies that the sub-program is not recursive, i.e. it does not call itself. By default, sub-programs have the capability to be recursive. At the time of writing, the compiler ignores this if present.

**EXIT SUB** causes program execution to return to the part of the program which called the sub-program. It is used to exit the sub-program before the **END SUB**.

Variables within a sub-program may be declared **LOCAL**, **SHARED** or **STATIC**. Please refer to these statements for detailed explanations and also the *User Manual : Concepts* chapter for more details of sub-programs.

User defined functions may not be a part of a sub-program definition.

- **See**

FUNCTION, STATIC, SHARED, LOCAL, EXIT, END, CALL.

- **Example**

```
CALL Fred(1,2)
```

```
.  
SUB Fred(One%,Two%)  
STATIC RecurDepth  
  INCR RecurDepth  
  IF RecurDepth > 4 THEN EXIT SUB  
  PRINT One% Two%  
  CALL Fred(One%*2,Two%*2)  
END SUB
```

Result:

```
1 2  
2 4  
4 8  
8 16
```

## **SWAP statement**

---

- **Syntax**

SWAP *1st\_variable*, *2nd\_variable*

- **Effect**

This statement causes the values of the two variables to be exchanged.

- **Comments**

The two variables must be of exactly the same type, and can be numeric or string.

- **See**

LSET, RSET.

- **Example**

```
A% = 1  
B% = 2  
SWAP A%,B%  
PRINT A%;B%
```

Result:

```
2 1
```

# SYSTAB function

## • Syntax

SYSTAB

## • Effect

Returns the address of an internal system table.

## • Comments

This is included mainly for ST BASIC compatibility, though can be useful for other reasons. The table consists of 13 ST BASIC compatible integers and various additional integers. The table should be accessed using

PEEKW(SYSTAB+offset)

but normal ST BASIC-style PEEKs will also work, though more slowly. The table contents are as follows:

|       |  |
|-------|--|
| 0     | Number of graphics planes  |
| 2     | <i>editor ghostline style = 2</i>  |
| 4     | <i>EDIT AES handle = 0</i>   |
| 6     | <i>LIST AES handle = 0</i>   |
| 8     | OUTPUT AES handle  |
| 10    | COMMAND AES handle = 0   |
| 12    | <i>EDIT open flag = 0</i>  |
| 14    | <i>LIST open flag = 0</i>  |
| 16    | OUTPUT open flag = 1   |
| 18    | COMMAND open flag = 0  |
| 20    | <i>graphics buffer, long integer = 0</i>   |
| 24    | GEMFLAG = 0  |
| 26    | x position of usable area  |
| 28    | y position of usable area  |
| 30    | x position of right hand of usable area  |
| 32    | y position of bottom of usable area  |
| 34    | copy of workstation VDI handle   |
| 36    | character pixel width  |
| 38    | character pixel height   |
| 40-46 | workstation size in the form x,y,w,h   |
| 48    | version number in hex, e.g. &h0123=1.23  |
| 50    | assumed screen address for PUT and GET - you can POKEL this with an alternate screen address |
| 54    | long integer pointer to the current base page  |
| 58-64 | x,y,w,h required to open a full sized window for use with WINDOW OPEN                        |
| 66    | byte the currency symbol for \$ in PRINT USING.  |
| 67    | If non-zero then sequential write files have a Ctrl-Z appended to them when they are closed. |

|    |   |
|----|---|
| 68 | The size of the LINE INPUT# buffer as a long word.                                |
| 72 | Top margin of GEM windows when using the PRINT statement (default 1) as a word    |
| 74 | bottom margin of GEM windows when using the PRINT statement (default 1) as a word |

Those shown in italics are never changed, but have fixed values for compatibility with ST BASIC.

The five integers starting at offset 26 return information about the current output window and VDI handle - this is a HiSoft BASIC extension useful for converting between screen co-ordinates and window co-ordinates.

TOS programs only have access to SYSTAB+67 and SYSTAB+68 - the other entries are not initialised.

## • See

PEEKL, PEEKW, PEEKB, WINDOW OPEN, LINE INPUT#, PRINT USING.

## • Example

```
DO
  IF MOUSE(2)=1 THEN
    x%=MOUSE(0)-PEEKW(SYSTAB+26)
    y%=MOUSE(1)-PEEKW(SYSTAB+28)
    PSET (x%,y%)
  END IF
LOOP UNTIL MOUSE(2) AND 2 'until right button
```

## SYSTEM statement

### • Syntax

SYSTEM [*return\_code*]

### • Effect

This statement causes program execution to end, closes all files, and returns to the operating system.

### • Comments

This statement does not issue a Program terminated - press any key message unlike STOP.

The optional parameter is a GEMDOS return code for passing back to the shell that has called the program.

SYSTEM without a parameter is equivalent to STOP -1.

### • See

END, STOP.

### • Example

```
IF UCASE$(a$)="QUIT" THEN SYSTEM
SYSTEM -33 'GEMDOS File not found error
```

## TAB function

### • Syntax

TAB(*n*)

### • Effect

This function causes the print position to move to the *n*th column.

### • Comments

If the current print position is already beyond *n*, then the print position will move to the *n*th column on the next line. If *n* is greater than the output width, the print position is moved to *n* MOD WIDTH. If *n* is less than one, the print position will become 1. TAB may only be used in PRINT and LPRINT statements.

### • See

PRINT, LPRINT, SPC.

### • Example

```
'80 column screen
PRINT TAB(-10) "minus ten"
PRINT TAB(175) "one-hundred-and-seventy-five"
PRINT TAB(10) "ten"
```

Result:

```
minus ten           one-hundred-and-seventy-five
                    ten
```



## TAN function

---

- **Syntax**

TAN(*numeric\_expression*)

- **Effect**

This function returns the tangent of *numeric\_expression* which must be in radians.

- **Comments**

The tangent is normally calculated with single-precision. If *numeric\_expression* is double-precision, the value returned will be calculated with double-precision.

- **See**

ATN, SIN, COS.

- **Example**

PRINT TAN(0.5)

## TIME\$ function

---

- **Syntax**

TIME\$

- **Effect**

This function returns the current system time.

- **Comments**

A string of the following format is returned:

*hh:mm:ss*    *hh* is the hours in 24hr format (00-23)  
                  *mm* is the minutes (00-59)  
                  *ss* is the seconds (00-59)

Note that the ST's internal clock only gives the time to the nearest 2-second increment.

- **See**

DATE\$, Tgettime GEMDOS call.

- **Example**

PRINT TIME\$

Result:

20:43:56

'it's 8:43 pm

## **TIME\$ statement**

---

- **Syntax**

TIME\$ = *string\_expression*

- **Effect**

The system time is set.

- **Comments**

*string\_expression* can have one of the following formats:

|                 |                                     |
|-----------------|-------------------------------------|
| <i>hh</i>       | sets only the hours                 |
| <i>hh:mm</i>    | sets the hours and minutes          |
| <i>hh:mm:ss</i> | sets the hours, minutes and seconds |

A 24hr clock is used, so 11pm is 23:00:00.

To read the time, use the TIME\$ function.

- **See**

DATE\$, Tsettime GEMDOS call.

- **Example**

TIME\$ = "19:30:00" '7:30pm

## **TIMER function**

---

- **Syntax**

TIMER

- **Effect**

This function returns an internal timer count in seconds as a single-precision floating point number.

- **Comments**

This function can be used to provide a seed for RANDOMIZE. It returns using 200ths of a second precision. It can be used for timing programs, for example.

There is no actual meaning to this number - it is merely an increasing number.

- **See**

TIME\$.

- **Example**

```
t!=TIMER
call Main
t!=TIMER-t!
PRINT "It took";t!;"seconds"
```

## TRON,TROFF statements

---

- **Syntax**

TRON  
TROFF

- **Effect**

These statements allow tracing of a program by line number as it runs.

- **Comments**

After a TRON statement has been executed all program line numbers are printed on the screen as execution occurs. It can be switched off with TROFF.

Line numbers are printed within square brackets.

The use of TRON in a program considerably slows down program execution, even after a TROFF statement. For this reason all TRONs should be removed from a program once it has been debugged for maximum performance.

TRON and TROFF may not be used in desk accessories.

- **Example**

```
1000 TRON
1010 PRINT "In problem area"
1020 INPUT a
1030 PRINT "Input";a,"Result";fnTest(a)
1040 TROFF
```

## UBOUND function

---

- **Syntax**

UBOUND(array\_name[,dimension])

- **Effect**

This function returns the largest available subscript of the array (and optionally the dimension) specified.

- **Comments**

*dimension* specifies the dimension of the array for which the upper bound is to be returned. If *dimension* is not specified the default is 1.

- **See**

LBOUND, DIM.

- **Example**

```
DIM An_array%(10,20,30)
PRINT UBOUND(An_array%,2)
```

Result:

20

## UCASE\$ function

---

- **Syntax**

UCASE\$(*string\_expression*)

- **Effect**

This function returns a string with every alphabetic character in upper case.

- **Comments**

Any characters in the range 'a' to 'z' inclusive are converted to 'A' to 'Z' as required. All other characters including accented letters are left alone.

- **See**

LCASE\$.

- **Example**

```
' the array must be passed by reference
' 1 defines the number of dimensions
FUNCTION find(BYVAL a$,array$(1))
STATIC i,f
f=0
FOR i=LBOUND(array$) TO UBOUND(array$)
IF UCASE$(f$)=array$(i) THEN f=i: EXIT FOR
NEXT i
find=f
END FUNCTION
```

## VAL function

---

- **Syntax**

VAL(*string\_expression*)

- **Effect**

This function returns the numeric value of *string\_expression*.

- **Comments**

This function parses *string\_expression* looking for a sequence of characters that can be interpreted as a number. VAL will stop reading the string upon finding the first character that cannot be recognized as a number. Leading blanks, tabs, and CR-LFs are ignored. VAL returns a double-precision number but it can, of course be assigned to a variable of less precision.

- **See**

STR\$.

- **Example**

```
addr$=" 26 Church Lane"
num%=VAL(addr$) 'will be 26
```

## VARPTR function

### • Syntax

VARPTR(*variable\_name*)

or

VARPTR(*#channel*)

### • Effect

This function returns the address in memory of the variable *variable\_name*, or of the input/output buffer of the *channel*.

### • Comments

The address returned is a long integer. VARPTR may be used on any numeric variable type; use SADD to find the address of strings. To find the address of an array, specify the first element in the array.

Be careful to use VARPTR just before the returned address is to be used. Local numeric variables and all strings and arrays may not remain in the same place for the duration of the program.

VARPTR with a channel number as a parameter returns the address of the input/ output buffer for an already OPENed channel. If the channel is not opened then 0 is returned. You can also use this to find whether a HiSoft BASIC window is open; use a channel number of 257+i for window i. This will return non-zero if the window is open; zero if it is not.

VARPTR will always return an even address.

The keyword VARPTR is also used in function definitions to denote variable parameters (as opposed to value parameters).

### • See

SADD, VARPTRS.

### • Example

```
Number% = 7
Addr& = VARPTR(Number%)
PRINT HEX$(Addr&)
```

Result:

6A558

## VARPTRS function

### • Syntax

VARPTRS(*subname*)

### • Effect

This function returns the address in memory of the sub-program *subname*.

### • Comments

The address returned is a long integer. This can be used in conjunction with CALLS. The sub-program should have no parameters.

*Note:* this is for advanced programmers only.

### • Example

```
IF from_disk THEN
    read_routine&=VARPTRS(disk_read)
ELSE
    read_routine&=VARPTRS(ram_read)
END IF
```

## WAVE statement

### • Syntax

WAVE *enable*[,*envelope*[,*shape*[,*period*[,*delay*]]]]

### • Effect

Controls the waveforms used by the SOUND statement.

### • Comments











*enable* is the mixer register of the sound generator and is a bitwise value as follows:

| bit | action  |
|-----|---|
| 0   | 0=enable tone on channel A, 1=disable tone on channel A   |
| 1   | 0=enable tone on channel B, 1=disable tone on channel B   |
| 2   | 0=enable tone on channel C, 1=disable tone on channel C   |
| 3   | 0=enable noise on channel A, 1=disable noise on channel A |
| 4   | 0=enable noise on channel B, 1=disable noise on channel B |
| 5   | 0=enable noise on channel C, 1=disable noise on channel C |

*envelope* is used to enable the envelope generator for the different channels. This is a bit organised value as follows:

| bit | action  |
|-----|---|
| 0   | 0=disable envelope for channel A, 1=enable envelope for channel A |
| 1   | 0=disable envelope for channel B, 1=enable envelope for channel B |
| 2   | 0=disable envelope for channel C, 1=enable envelope for channel C |

*shape* is the envelope shape as follows

| shape value | wave form   |
|-------------|---|
| 0 - 3       |  |
| 4 - 7       |  |
| 8           |  |
| 9           |  |
| 10          |  |
| 11          |  |
| 12          |  |
| 13          |  |
| 14          |  |
| 15          |  |

*period* sets the period of the envelope (from 1 to 65535) and *delay* determines the time in 50ths of a second before the command finishes.

### • See

SOUND.

### • Example

```
SOUND 1,15,10,3,20
WAVE 1,1,12,10,20
SOUND 1,15,10,3,20
SOUND 1,0
```

## WHILE...WEND statement

### • Syntax

WHILE *condition*

.

[*statements*]

.

WEND

### • Effect

A series of statements are executed in a loop until *condition* becomes false.

### • Comments

While *condition* is true, the statements between WHILE and WEND are executed.

WHILE...WENDs may be nested; a WEND will match the last WHILE.

Be careful not to begin a WHILE...WEND loop from within it (e.g. due to a GOTO).

DO...LOOP and REPEAT loops are more general versions of the while loop.

### • See

DO...LOOP, REPEAT.

### • Example

```
A%=0
WHILE A%<10
  INCR A%
  PRINT A%;
WEND
```

Result:

1 2 3 4 5 6 7 8 9 10

## WIDTH statement

### • Syntax

WIDTH [#*channel\_number*,] *width* [,*tab\_setting*]

WIDTH LPRINT *width* [,*tab\_setting*]

### • Effect

These statements assign a line width to the specified file, screen or printer.

### • Comments

The following parameters are valid for this statement:

|  |   |
|--|---|
| <i>width</i>                           | This sets the width for output to the screen.   |
| # <i>channel_number</i> , <i>width</i> | This sets the width of output to <i>channel_number</i> to the specified width. Note that the # is compulsory. |
| <i>tab_setting</i>                     | sets the length of a tab; default is 14.  |

Specifying LPRINT controls the *width* and *tab\_length* of any output to the printer.

If a *width* of 255 is specified then the width is set to 'infinite'; that is BASIC never inserts CR-LFs during output.

### • Example

|                   |   |
|-------------------|---|
| WIDTH 80          | 'sets an 80 column screen                             |
| WIDTH LPRINT 40,8 | 'set printer output to 40 columns and tab length of 8 |

# WINDOW statements

## • Syntax

WINDOW OPEN *id*,*title\_string*,*xpos*,*ypos*,*width*,*height*,*type*  
WINDOW CLOSE [*id*]  
WINDOW OUTPUT [*id*]  
WINDOW FULLW [*id*]  
WINDOW NAME *id*[,*title\_string*][,*status\_string*]  
WINDOW LOCATE *id*,*xpos*,*ypos*,*width*,*height*  
WINDOW ON  
WINDOW OFF  
WINDOW CONTRL *id*,*control*,*value*  
WINDOW READ *id*,*control*,*result*  
WINDOW GET *id*,*rectangle*,*xpos*,*ypos*,*width*,*height*  
WINDOW SYSTEM  
WINDOW STOP

## • Effect

Control multi-window screen output.

## • Comments

There are many versions of the WINDOW command to help use and control multi-window output via GEM.

Most commands require a window *id*, which is a number used to identify each window, ranging from 0 to 7 inclusive. By default when a GEM program starts it opens window *id* 2 to be the whole screen size except for the menu bar. All PRINT, INPUT and graphics commands use this window, known as the *current window*, until directed otherwise.

Note that a window *id* is a method used by BASIC to refer to windows - *ids* should not be passed to the AES without conversion to proper AES handles, using WINDOW READ.

Now the individual window commands are discussed in detail.

### WINDOW OPEN *id*,*title\_string*,*xpos*,*ypos*,*width*,*height*,*type*

This opens a new window, after closing any existing window with the given *id*. The title of the window is specified by the *title\_string*, and *xpos*, *ypos*, *width* and *height* specify its position relative to the top left of the screen and its size.

The *type* parameter determines what sort of window is produced, and should be made up by adding together the following numbers for the attributes required:

|      |         |   |
|------|---------|---|
| 1    | (&h1)   | Title bar   |
| 2    | (&h2)   | Close box   |
| 4    | (&h4)   | Full box  |
| 8    | (&h8)   | Move box, allowing it to be dragged around the screen |
| 16   | (&h10)  | Information line, empty initially (see WINDOW NAME)   |
| 32   | (&h20)  | Size box, allowing it to be re-sized                  |
| 64   | (&h40)  | Up arrow  |
| 128  | (&h80)  | Down arrow  |
| 256  | (&h100) | Vertical slider                                       |
| 512  | (&h200) | Left arrow  |
| 1024 | (&h400) | Right arrow   |
| 2048 | (&h800) | Horizontal slider                                     |

The newly opened window will be at the front and made the current window.

### WINDOW CLOSE [*id*]

This closes the given window. If this was the current window then any attempt to PRINT or otherwise use it will cause an error. If *id* is omitted it defaults to 2.

### WINDOW OUTPUT [*id*]

This makes the given window the current one. All PRINT, INPUT and graphics commands will go to it. If *id* is omitted it defaults to 2.

### WINDOW FULLW [*id*]

This makes the given window bigger than the complete size of the screen such that the usable area is the whole screen. The window covers the menu bar rendering it unusable. This can be useful for full screen graphics. If *id* is omitted it defaults to 2.

### WINDOW NAME *id* , [*title\_string*] [,*status\_string*]

This lets you change the title and/or the status line on the given window. If either string is omitted then that attribute will be left alone.

### WINDOW LOCATE *id*,*xpos*,*ypos*,*width*,*height*

This lets you move and re-size windows around the screen.

### WINDOW ON

This command lets the run-time system take control of all VDI clipping and messages, and is the default state.

### WINDOW OFF

This command stops the run-time system from doing clipping and message absorption, and is intended for more advanced programmers who wish to use the AES and VDI routines for screen output and message handling.



### WINDOW CONTRL *id,control,value*

This command lets you set the values of the different window controls, such as scroll bars, and change which window is at the front. The *control* values should be one of the following:

|   | Control                    | Value                      |
|---|----------------------------|----------------------------|
| 0 | horizontal slider position | 1=leftmost, 1000=rightmost |
| 1 | vertical slider position   | 1=top, 1000=bottom         |
| 2 | horizontal slider size     | 1 to 1000                  |
| 3 | vertical slider size       | 1 to 1000                  |
| 4 | set front window           | ignored - uses <i>id</i>   |

### WINDOW READ *id,control,result*

This command lets you read the values of the different window controls, such as scroll bars. The *result* argument must be an integer variable, into which the result will be placed.

|   | Control                    | Result  |
|---|----------------------------|---|
| 0 | horizontal slider position | 1=leftmost, 1000=rightmost  |
| 1 | vertical slider position   | 1=top, 1000=bottom  |
| 2 | horizontal slider size     | 1 to 1000   |
| 3 | vertical slider size       | 1 to 1000   |
| 4 | read front window          | window <i>id</i> or -1 if not a BASIC window, -2 if no windows, ignores <i>id</i> parameter |
| 5 | read AES handle            | AES handle  |
| 6 | convert from AES handle    | window <i>id</i> , or -1 if not a BASIC window  |

### WINDOW GET *id,rectangle,xpos,ypos,width,height*

This command lets you read the co-ordinates of the various rectangle which make up a GEM window, intended for advanced programmers. The variables *xpos,ypos,width* and *height* must all be integer variables, into which the results are placed.

#### Rectangle

|   |  |
|---|--|
| 0 | usable area of window                  |
| 1 | total size of window including borders |
| 2 | co-ordinates of window at maximum size |
| 3 | first rectangle in window list         |
| 4 | next rectangle in window list          |

### WINDOW SYSTEM

This command causes the GEMVDI workstation to be opened and the window system to be initialised. The appropriate items in the SYSTAB table are also initialised. WINDOW SYSTEM may only be used in desk accessories.

### WINDOW STOP

This command causes the GEMVDI workstation to be closed; any open windows should be closed before calling this. This may only be used in desk accessories.

#### • See

PRINT, WRITE, PSET, GEMAES window functions, GEMVDI library.

#### • Example

```
' Get the position, width and height of the free  
' window that HiSoft BASIC gives you
```

```
WINDOW GET 2,1,wx,wy,ww,wh
```

```
' Re-locate and name the free window you get  
' when running a HiSoft BASIC GEM program.  
' This always has id number 2.
```

```
WINDOW LOCATE 2,0,25,320,200,wh\2  
WINDOW NAME 2," Top Window "  
COLOR 1,1,1,5,2  
PELLIPSE 320,WHEIGHT\2,200,WHEIGHT\4  
WINDOW OPEN 1," Another Window ",  
0,25+wh\2,640,wh\2,1+512+1024+2048  
PRINT "Hello"  
WINDOW CONTRL 1,0,500  
WINDOW CONTRL 1,2,100
```

## WRITE statement

### • Syntax

WRITE [*expression\_list*]

### • Effect

This statement prints the data specified in *expression\_list* to the screen.

### • Comments

This statement is similar to PRINT. If *expression\_list* is omitted, a CR-LF will be printed to the screen. All expressions must be separated by commas. These commas are printed as well. Semi-colon may be used instead of comma; this also outputs a comma.

Strings printed using WRITE are quoted, and numbers printed using WRITE do not have leading or trailing spaces, unlike PRINT.

A CR-LF is always output at the end of a WRITE statement.

### • See

PRINT, WRITE#, INPUT.

### • Example

```
A% = 24
B$ = "STRING!"
PRINT A%,B$
WRITE A%,B$
```

Result:

```
24      STRING!
24, "STRING!"
```

## WRITE# statement

### • Syntax

WRITE #*channel\_number*, *expression\_list*

### • Effect

This statement writes the data specified by *expression\_list* to a sequential file.

### • Comments

WRITE# differs from PRINT# in the same ways as WRITE differs from PRINT. WRITE# has the advantage that data can be read back from the file with the equivalent INPUT# statement; this is not normally the case for PRINT.

For further details, please consult the PRINT# and WRITE entries.

### • See

PRINT#, WRITE, INPUT#.

### • Example

```
OPEN f$ FOR OUTPUT AS #1
FOR i=1 TO 10
    WRITE #1,i,CHR$(i+"a%")
NEXT i
CLOSE #1
OPEN f$ FOR INPUT AS #1
WHILE NOT EOF(1)
    INPUT #1,a,a$
    PRINT a,a$
WEND
CLOSE #1
```

# Chapter 2

## Library Reference

---

### Introduction

---

HiSoft BASIC is supplied with several *libraries*, a collection of functions and/or sub-programs callable, with ease, from BASIC. The file HBASIC.LIB contains, apart from the normal run-times, five additional libraries:

|        |                                  |
|--------|----------------------------------|
| BIOS   | Basic Input/Output (I/O)         |
| XBIOS  | Extended Basic I/O               |
| GEMDOS | The Operating System             |
| GEMVDI | Virtual Device Interface         |
| GEMAES | Application Environment Services |

If you want to use any particular library you have to specify its name via the LIBRARY statement, then you can use all of its features.

For example, there is an XBIOS function to find out where the screen is in memory, so if you wanted to save a copy of it to disk you could use this simple program:

```
LIBRARY "xbios"
```

```
CIRCLE 100,100,100
```

```
PRINT "Hello"
```

```
BSAVE "SCREEN.DMP",logbase&,32000
```

The first line says *I want to use the XBIOS*, while on the last line you can see `logbase&`, which is a function defined not in your program, like normal functions, but in the library itself.

When using the supplied libraries you will often need to pass an *address* as a parameter. This can be done using the VARPTR function, or SADD in the case of strings, where required.

There are a few functions and subroutines which are absolutely straightforward to use, such as `logbase&` mentioned previously. However a good number of the BIOS, XBIOS and GEMDOS calls can cause complete havoc to the system. If you don't fully understand a call before using it, the machine is quite likely to crash, so always save any vital data on a permanent disk before trying a new one out.

Space does not permit a complete description of every one of the system calls in this document, so you are referred to the books in the *Bibliography* in the *User Manual* for further details.

## Important Note

Please note that we have listed all the sub-programs and functions in a form that shows their definition. This means that each heading begins with the word SUB or FUNCTION depending on the type of the library entry - obviously you do not use the word SUB or FUNCTION when you call the routine.

Also, the routine's parameters are shown as they would be defined; if a parameter is preceded by the word BYVAL then it is a value parameter, otherwise it is a variable parameter, which is the default for SUBs and FUNCTIONS. Again, you must not type the word BYVAL when you call the routine in question - it is there just to show you what type of parameter the routine accepts.

## The Libraries

### The GEMDOS Library

#### SUB pterm0

This statement makes the old GEMDOS call to terminate a program. It should never be used to terminate a BASIC program.

#### FUNCTION cconin&

This function returns one character from the keyboard. The high-word contains the scancode of the key pressed, the low-word contains the ASCII code.

#### SUB cconout(BYVAL char%)

A statement that prints the ASCII character char% to the console.

#### FUNCTION cauxin%

This function returns a character from the serial port AUX: or a TOS error number.

#### FUNCTION cauxout%(BYVAL chr%)

Writes a character to AUX: and returns a (negative) TOS error number if it fails. A function.

#### FUNCTION cprnout%(BYVAL chr%)

This function writes a character to PRN: and returns a (negative) TOS error number if it fails.

#### FUNCTION crawio%(BYVAL chr%)

A function that returns a character from the standard input if chr%=&HFF; otherwise a character is written to the standard output.

#### FUNCTION crawcin%

This function returns a raw character from the standard input without echo.

#### FUNCTION cnecin%

Returns a raw character from the standard input without echo; Ctrl-C, Ctrl-S and Ctrl-Q are interpreted and have effect. A function.

#### SUB cconws(BYVAL string\$)

This statement causes string\$ to be written to the standard output.

#### SUB cconrs (BYVAL buffer%)

A statement that causes a line to be read from the standard input into buffer%. The array need not be longer than 256 bytes (i.e. 128 elements), as the maximum string length returned from GEMDOS is 255 bytes; the maximum number of characters that can be typed in is determined by the size of the array. Ctrl-C, Ctrl-S and Ctrl-Q are interpreted and have effect.

#### FUNCTION cconis%

This function returns -1 if a character is available from the standard input, 0 is returned otherwise.

#### FUNCTION dsetdrv%(BYVAL driveno%)

Sets the GEMDOS current drive; 0 for A:, 1 for B: etc. The system drive bitmap is returned (bit 0 set means drive A: is available etc.). A function.

#### FUNCTION cconos%

Function returns -1 if the console is ready to receive a character, 0 if not.

### FUNCTION cprnos%

Returns -1 if the printer port is ready to receive a character, 0 if it isn't.

### FUNCTION cauxis%

A function that returns -1 if a byte is available from the serial port, 0 if not.

### FUNCTION cauxos%

Returns -1 if the serial port is ready to receive a character, 0 if it isn't.

### FUNCTION maddalt&(BYVAL start&, BYVAL size&)

This is used to inform the operating system of the presence of alternative memory in the system - only for use by VME RAM on machines that support this and is available on TOS 2 and above only. 0 is returned if the function succeeds; otherwise an error code is given.

### FUNCTION dgetdrv%

This function returns the GEMDOS current drive, 0 for A:, 1 for B: etc.

### SUB fsetdta(BYVAL buffer&)

A statement that sets the disk transfer address to the 44-byte buffer pointed to by *buffer&*. If you use *fsetdta*, be sure to restore the old DTA (which is returned by *fgetdta*). If you do not do this, the BASIC file system will become confused.

### FUNCTION super&(BYVAL address&)

This function causes the processor to go into supervisor mode or to come out of it. Upon entering supervisor mode, *address&* is the value of the new supervisor stack pointer or 0 if the current user stack pointer should be used. The returned long integer is the old supervisor stack pointer which *must* be passed back to *super&* when leaving supervisor mode. When using *super&* to enter supervisor mode, make sure to leave supervisor mode properly as well. Remain in supervisor mode only as long as necessary.

### FUNCTION tgetdate%

A function that returns the system date in the format:

| Bits |                       |
|------|-----------------------|
| 0-4  | day 1-31              |
| 5-8  | month 1-12            |
| 9-15 | year 0-119 since 1980 |

### SUB tsetdate(BYVAL date%)

This statement sets the date using the format shown above.

### FUNCTION tgettime%

A function that returns the system time in the format:

| Bits  |                             |
|-------|-----------------------------|
| 0-4   | second 0-59, divided by two |
| 5-10  | minute 0-59                 |
| 11-15 | hour 0-23                   |

### SUB tsettime(BYVAL time%)

This statement sets the system time using the format shown above.

### FUNCTION fgetdta&

This function returns the current disk transfer address.

### FUNCTION sversion%

A function that returns the TOS version number.

### SUB ptermres(BYVAL ret%, BYVAL howmuch&)

This statement calls the GEMDOS terminate and stay resident routine. *ret%* is the return code, *howmuch&* is the number of bytes to keep resident in memory including the basepage. Like any of the other terminate calls, this must not be used from a BASIC program.

### SUB dfree(BYVAL buffer&(), BYVAL drive%)

Returns allocation information about *drive%* (0=current drive, 1=A:, 2=B: etc.). The array *buffer&()* will contain four long integers in the following order: free clusters, total clusters, bytes per sector, and sectors per cluster. A statement.

### FUNCTION dcreate%(BYVAL dirname\$)

This function creates a directory. The value returned is a negative TOS error number if creation failed.

### FUNCTION ddelete%(BYVAL dirname\$)

Deletes a directory. The value returned is a negative TOS error number if deletion failed. A function.

### FUNCTION dsetpath%(BYVAL dirname\$)

This function sets the current directory.

## FUNCTION create%(BYVAL filename\$, BYVAL attributes%)

A function that creates a file with the given pathname. Bits in attributes% can be set as shown below:

| Bit | Attribute    |
|-----|--------------|
| 0   | Read Only    |
| 1   | Hidden       |
| 2   | System       |
| 3   | Volume label |

If a file has both hidden *and* system attributes, it will become visible from the Desktop; either Hidden *or* System will render the file invisible. The integer returned is the file handle or a negative TOS error number.

## FUNCTION fopen%(BYVAL filename\$, BYVAL mode%)

This function opens the file specified. mode% can be 0 for read, 1 for write, and 2 for read and write. The integer returned is the file handle or a negative TOS error number.

## FUNCTION fclose%(BYVAL handle%)

This function closes a file. handle% is the positive return value from either fcreate% or fopen%.

## FUNCTION fread%(BYVAL handle%, BYVAL count%, BYVAL buffer%)

This function reads count% bytes into memory pointed to by buffer%. handle% is the file's handle. The value returned is the actual number of bytes read unless negative, in which case it is a TOS error number.

## FUNCTION fwrite%(BYVAL handle%, BYVAL count%, BYVAL buffer%)

A function that writes count% bytes from memory pointed to by buffer%. handle% is the file's handle. The value returned is the actual number of bytes written unless negative, in which case it is a TOS error number.

## FUNCTION fdelete%(BYVAL filename\$)

Deletes the specified file. 0 is returned if successful, a negative value if a TOS error has occurred. A function.

## FUNCTION fseek%(BYVAL offset%, BYVAL handle%, BYVAL mode%)

This function seeks within a file. offset% is the number of bytes to seek by. The result is the new file position mode% can be:

| Mode |                             |
|------|-----------------------------|
| 0    | seek from beginning of file |
| 1    | seek from current position  |
| 2    | from end of file            |

## FUNCTION fattrib%(BYVAL filename\$, BYVAL mode%, BYVAL attribute%)

This function sets a file's attributes if mode% is 1, or reads them if mode% is 0. The attribute bits are as follows:

| Bit | Attribute   |
|-----|---|
| 0   | Read Only   |
| 1   | Hidden  |
| 2   | System  |
| 3   | Volume Label                                      |
| 4   | Subdirectory                                      |
| 5   | Archive bit, set when file is written to & closed |

## FUNCTION mxalloc%(BYVAL size%, BYVAL mode%)

This is the general call to allocate memory that is available in TOS 2 and above. size% is the number of bytes to allocate and the function returns the address of the area allocated or an error code if it failed. mode% indicates the type of memory to return:

- 0 System (ST) RAM only
- 1 Alternative (TT) RAM only
- 2 System (ST) RAM if available otherwise TT RAM
- 3 Alternative (TT) RAM if available otherwise ST RAM.

Thus you would use 0 if the memory is going to be used as a physical screen address. Generally 3 is used for memory that you would like to be accessed as quickly as possible.

## FUNCTION fdup%(BYVAL handle%)

A function that returns a duplicate handle that refers to the same file or device as handle%.

## SUB fforce(BYVAL stdhandle%, BYVAL nonstdhandle%)

A statement that forces stdhandle% to point to same file or device as nonstdhandle%.



### SUB dgetpath(BYVAL buffer&, BYVAL drive%)

Returns the current directory in the buffer pointed to by `buffer&`. The buffer must be at least 64 bytes long. `drive%` is 0 for the current drive, 1 for A: etc. A statement.

### FUNCTION malloc&(BYVAL howmuch&)

This function requests `howmuch&` memory from the operating system. The return is either an address on a word boundary of `howmuch&` bytes or 0 if allocation failed. If `howmuch&` is -1, then the amount of available system memory is returned. All memory allocated with `malloc&` must be de-allocated with `mfree&`.

### FUNCTION mfree%(BYVAL where&)

Function that returns memory allocated by `malloc&` to the operating system. `where&` is the address that was returned by `malloc&`. A negative TOS error number is returned if an error occurred, otherwise the return value is 0.

### FUNCTION mshrink%(BYVAL where&, BYVAL howmuch&)

This function returns memory to the operating system. `where&` is the beginning of the memory block, `howmuch&` is the amount of memory to retain. A negative TOS error number is returned upon failure.

### FUNCTION pexec%(BYVAL mode%, BYVAL filename\$, BYVAL commandline\$, BYVAL environment\$)

Executes the specified program with the specified parameters. The return value is either a TOS error or the return value passed by the program executed upon termination. This function offers the ability to execute a program and then return. The BASIC RUN and CHAIN statements do not return to the calling program. There is a potentially major problem when using this function: the operating system must have enough memory to run the program (i.e. load it into memory) and there must be enough memory for the called program to run. Consult *Appendix B* and *Chapter 3* of the *User Manual* for details on the KEEP and LEAVE memory allocation options.

### SUB pterm(BYVAL return%)

This is another of GEMDOS's terminate calls. `return%` will be passed back to the process that executed the program. This call must not be used from BASIC.

### FUNCTION fsfirst%(BYVAL filename\$, BYVAL attributes%)

This function searches for the occurrence of `filename$` which has the specified `attributes%`. `filename$` may contain wildcards as long as they are not part of the pathname. 0 is returned if the file was found, -33 if not. The DTA is filled with information about the file:

| Bytes |   |
|-------|---|
| 0-20  | GEMDOS internal information                 |
| 21    | File attributes                             |
| 22-23 | File time stamp                             |
| 24-25 | File date stamp                             |
| 26-29 | File size (long integer)                    |
| 30-43 | Name and extension of file, null terminated |

### FUNCTION fsnext%

This function returns the next occurrence of the file specified by `fsfirst%`. It must therefore not be used before a `fsfirst%` call. To be completely safe, do not make any other GEMDOS calls between calls to `fsfirst%` and any calls to `fsnext%`.

### FUNCTION frename%(BYVAL oldfile\$, BYVAL newfile\$)

Renames `oldfile$` to `newfile$`. A negative return value is a TOS error. A function.

### SUB fdatetime(BYVAL handle%, BYVAL buffer&, BYVAL set%)

This statement gets or sets a file's time and date stamp. `handle%` is the file's handle and `buffer&` is a pointer to a buffer which contains/will receive the time and date stamp. If `set%` is 0, the stamp will be returned, if `set%` is 1, the stamp is set.

## The BIOS library

### SUB getmpb(BYVAL mpb&())

`mpb&()` is an array in which three long integers are returned.

This function returns three 32-bit addresses in the array. These addresses are pointers to the memory free list, memory allocated list and the roving pointer.

### FUNCTION bconstat%(BYVAL dev%)

dev% is a short integer that specifies the device to be checked.

This function returns the status of the device. 0 is returned if no characters are available, -1 is returned if at least one character is available. dev% may be

|   |                        |
|---|------------------------|
| 0 | PRT: the parallel port |
| 1 | AUX: the serial port   |
| 2 | CON: the console       |
| 3 | MIDI port              |
| 4 | Keyboard port          |

### FUNCTION bconin%(BYVAL dev%)

dev% is the device from which the character is returned; the values for dev% are the same as for bconstat%.

This function returns a character from the specified device in the low word. If the character is returned from CON: then the high word contains the scancode of the key that was pressed.

### FUNCTION bconout%(BYVAL dev%, BYVAL char%)

dev% is the device to which the character is written. char% is the integer ASCII value of the character to be written.

This function writes one character to the specified device. If the device is PRT:, -1 is returned if the character could be written, 0 if the call failed. The function returns only after the character was written or if the device timed out.

### FUNCTION rwabs%(BYVAL rwflag%, BYVAL buffer%, BYVAL count%, BYVAL recno%, BYVAL dev%)

rwflag% can be

|   |                            |
|---|----------------------------|
| 0 | read                       |
| 1 | write                      |
| 2 | read; ignore media change  |
| 3 | write; ignore media change |

buffer% is the address of the buffer. count% is the number of sectors to be read or written. recno% is the logical sector number of the first sector to be read from or written to. dev% is the drive: 0 for drive A:, 1 for drive B: etc.

This function reads from or writes to sectors of the specified drive. 0 is returned if no error occurred; any negative values are TOS error numbers.

### FUNCTION setexc%(BYVAL number%)

number% is the vector number to be read.

This function returns the specified 68000 vector. It is not possible to set a vector using this function.

### FUNCTION tickcal%

This function returns the time in milliseconds between two system-timer calls.

### FUNCTION getbpb%(BYVAL dev%)

dev% is the drive; 0 for drive A:, 1 for drive B: etc.

This function returns a pointer to the BIOS parameter block of the specified drive. The block is defined as follows:

| Offset | Meaning                            |
|--------|------------------------------------|
| 0      | bytes per sector                   |
| 2      | sectors per cluster                |
| 4      | bytes per cluster                  |
| 6      | size of root directory in sectors  |
| 8      | size of FAT in sectors             |
| 10     | first sector of the second FAT     |
| 12     | first sector of data               |
| 14     | number of data sectors on the disk |
| 16     | various flags                      |

### FUNCTION bcostat%(BYVAL dev%)

dev% is the device to be checked. Valid device numbers are the same as for bconstat%. This function returns -1 if the device is ready to receive data, 0 if it isn't.

### FUNCTION mediach%(BYVAL dev%)

dev% is 0 for drive A:, 1 for drive B: etc.

This function returns whether or not the media has been changed in the drive specified. If 0 is returned, the media has definitely not changed. If 1 is returned, status is unsure. If 2 is returned, media has definitely changed.

### FUNCTION drvmap%

This function returns a bitmap of the active drives. Bit 0 corresponds to drive A:, bit 1 to drive B: etc.

### FUNCTION kbshift%(BYVAL mode%)

If mode% is negative, the status will be returned; if it is non-zero, the bits will be set.

This function returns or sets the status of the Shift keys in a bitmap.



The bits correspond to the following keys:

|   |                 |
|---|-----------------|
| 0 | right Shift key |
| 1 | left Shift key  |
| 2 | Ctrl key        |
| 3 | Alt key         |
| 4 | Caps Lock       |

The above is the only way to legally determine the status of the Caps Lock key. All other keys can also be tested by using the AES or VDI.

**SUB BLOCKMOVE**(BYVAL source&, BYVAL dest&,  
BYVAL length&)

This moves length& bytes of memory from address source& to dest&. BLOCKMOVE isn't an actual XBIOS facility, but is often useful when using these routines.

## ***The XBIOS Library***

**SUB initmous**(BYVAL type%, BYVAL param&, BYVAL  
vec&)

A sub-program to initialise routines that are executed when the mouse is used.

**SUB ssbrk**(BYVAL amount%)

This call has no effect with ROM TOS.

**FUNCTION physbase&**

This function returns the the video-RAM's physical base address at the beginning of the next vertical blank.

**FUNCTION logbase&**

A function that returns the logical screen base address.

**FUNCTION getrez%**

This function returns the current screen resolution, where 0 is low-res, 1 is medium-res, and 2 is high-res (monochrome).

**SUB setscreen**(BYVAL logloc&, BYVAL physloc&,  
BYVAL rez%)

A statement that sets the logical screen location (logloc&), the physical screen location (physloc&), and the resolution (rez%). If any of these parameters is negative, the value is not changed. Beware that screen resolution should not be changed with this call after GEM has been initialised.

**SUB setpalette**(BYVAL palette%())

This statement sets the entire colour palette (colours 0 to 15). palette%(0) should contain the colour value for colour 0 in the palette etc.

**FUNCTION setcolor%**(BYVAL colournum%, BYVAL  
colour%)

Sets colournum% in the palette to colour%. The old value of the palette register is returned. If colour% is negative, the colour is not changed. A function.

**FUNCTION floprd%**(BYVAL buffer&, BYVAL  
dev%, BYVAL sect%, BYVAL trac%, BYVAL  
side%, BYVAL count%)

This function reads count% sectors from track trac%, side side% beginning at sector sect%, dev% is the drive number; A: is 0, B: is 1.

The data from the disk is read into the area in memory specified by buffer&. No checks are made that this is memory that can freely be used by the program. The long integer returned is the error number. If it is 0, then the call was completed without error; any negative value returned is a TOS error number.

**FUNCTION flopwr%**(BYVAL buffer&, BYVAL dev%,  
BYVAL sect%, BYVAL trac%, BYVAL side%,  
BYVAL count%)

This function has the same parameters as floprd%. The major difference between the two is that flopwr% writes the data from buffer& to disk.

FUNCTION flopfmt%(BYVAL buffer&, BYVAL  
filler&, BYVAL dev%, BYVAL sectrk%,  
BYVAL trac%, BYVAL side%, BYVAL  
interlv%)

This function formats a track on a disk. dev%, trac%, and side% are the same as in floprd% and flopwr%. buffer& needs to be a minimum 9k area of free memory, where the XBIOS can prepare the image of the track. sectrk% specifies the number of sectors per track; 9 is the TOS standard. interlv% is the sector interleave factor which should be specified as 1. If bad sectors are encountered a null terminated list of the bad sectors can be found in the buffer. filler& should normally be zero, or point to a skew table if used on a blitter ROM.

SUB midiws(BYVAL count%, BYVAL data\$)

This call writes data\$ to the MIDI port. count% is the number of bytes to write.

SUB mfpint(BYVAL intno%, BYVAL vector&)

This sub-program sets the mfp interrupt intno% to point to vector&. The old address is over-written.

FUNCTION iorec%(BYVAL devno%)

A function that returns a pointer to the input buffer record of the specified device. 0 for the serial port, 1 for the keyboard, and 2 for the MIDI port.

SUB rsconf(BYVAL speed%, BYVAL flowctl%, BYVAL  
ucr%, BYVAL rsr%, BYVAL tsr%, BYVAL  
scr%)

This sub-program sets the configuration of the serial port. If any of the parameters are -1, then that value is not changed.

FUNCTION keytbl%(BYVAL unshift&, BYVAL shift&,  
BYVAL capslok&)

This function sets the keyboard map to the values pointed at by the three parameters. Any parameter that is -1 will cause that respective table not to be changed. A pointer to the addresses of the three tables is returned. The new tables must be kept in memory that will not be corrupted, otherwise the keyboard layout will become intriguing.

FUNCTION random&

This function returns a 24-bit random number produced by the XBIOS random number generator.

SUB protobt(BYVAL buffer&, BYVAL serial&,  
BYVAL disktype%, BYVAL execflag%)

A sub-program that produces a disk boot-sector image in the 512-byte buffer&. serial& is the 24-bit serial number of the boot disk; if it is greater than or equal to &H1000000, the serial number will be random. disktype% can be

|   |                         |
|---|-------------------------|
| 0 | 40 tracks, single sided |
| 1 | 40 tracks, double sided |
| 2 | 80 tracks, single sided |
| 3 | 80 tracks, double sided |

If execflag% is 1, the boot sector will be made executable; if 0, it will be made non-executable.

If serial&, disktype%, or execflag% are -1, the values will not be changed from those already in the buffer.

FUNCTION flopver%(BYVAL buffer&, BYVAL dev%,  
BYVAL sect%, BYVAL trac%, BYVAL side%,  
BYVAL count%)

The parameters for this function are identical to those of floprd% and flopwr%. This function verifies one or more sectors by reading them. buffer& must point to a 1024 byte buffer. A null terminated list of the bad sectors is returned in the buffer.

FUNCTION scrdmp%

Causes a screen dump to be sent to the printer. If the dump occurs without an error, 0 is returned. Otherwise, a negative TOS error number is returned.

FUNCTION cursconf%(BYVAL function%, BYVAL  
operand%)

This function defines the cursor. function% may be

|   |                              |
|---|------------------------------|
| 0 | hide cursor                  |
| 1 | show cursor                  |
| 2 | set cursor to blink          |
| 3 | set cursor not to blink      |
| 4 | set cursor blink to operand% |
| 5 | return blink timer value     |

The blink rate depends on the video scan rate (70hz for monochrome). The unit of measure for operand% is one-half the cycle time.

### SUB settime(BYVAL datetime&)

This statement sets the system clock/calendar. *datetime&* contains time in the low word, date in the high word; both are in DOS format.

### FUNCTION gettime&

A function that returns the value of the system clock/calendar. The format is the same as for *settime*.

### SUB bioskeys

A statement that resets the keyboard map to the power-up setting. A keyboard initialised by *keytbl&* is reset by this call.

### SUB ikbdws(BYVAL count%, BYVAL data\$)

Writes *data\$* to the intelligent keyboard. *count%* is the number of bytes to write. A statement.

### SUB jdisint(BYVAL intno%)

A statement that disables interrupt number *intno%* on the 68901 mfp chip.

### SUB jenabint(BYVAL intno%)

Enables interrupt number *intno%* on the 68901 mfp chip. A statement.

### FUNCTION giaccess%(BYVAL data%, BYVAL regno%)

This function reads from or writes to a register on the sound chip. *data%* is a byte transferred to the chip, *regno%* is the register number to read; if this value is ORed with &H80 (i.e. bit 7 is set) the register is written to.

### SUB offgibit(BYVAL bitno%)

This statement atomically (interrupt mask set to 7) clears *bitno%* in the PORT A register of the sound chip.

### SUB ongibit(BYVAL bitno%)

Atomically sets *bitno%* in the PORT A register of the sound chip.

### SUB xbtimer(BYVAL timer%, BYVAL control%, BYVAL data%, BYVAL vector&)

A statement that sets up a timer of the 68901 mfp chip. *timer%* can be a number from one to four, corresponding to the timers A to D. *control%* is the timer's control register setting, *data%* is the byte put into the timer's data register; *vector&* is a pointer to the interrupt handler. Timer A is the only timer reserved for user applications, the other three are used by the system.

### FUNCTION dosound&(BYVAL pointer&)

Causes the sound daemon to execute the commands at *pointer&*. A function.

### FUNCTION setprt%(*config%*)

Sets/returns the printer configuration byte. If *config%* is -1 then the current byte is returned, otherwise it is set with the value of *config%*. The bits are defined as (the other bits of *config%* are reserved and must be cleared):

| Bit | If cleared     | If set           |
|-----|----------------|------------------|
| 0   | Dot matrix     | Daisy wheel      |
| 1   | Colour printer | Monochrome       |
| 2   | Atari printer  | Epson compatible |
| 3   | Draft mode     | Final mode       |
| 4   | Parallel port  | Serial port      |
| 5   | Form-feed      | Single-sheet     |

### FUNCTION kbdrvbase&

Returns a pointer to a list of addresses of various buffers and interrupt handlers. A function.

### FUNCTION kbrate%(BYVAL initial%, BYVAL repeat%)

This function gets and sets the keyboard repeat rate. *initial%* is the initial delay before repeat sets in; *repeat%* is the rate of repeat. The values are based on 50Hz system ticks. If a parameter is -1, it is not changed.

### SUB rtblk(BYVAL listadr&)

This is a primitive used by *scrddmp*. The parameter passed is a pointer to a list of addresses and values.

### SUB vsync

This statement returns after the next vblank interrupt. It can be used for synchronizing graphics with the vblank interrupt.

## SUB supexec(BYVAL pointer&)

This statement causes the routine at `pointer&` to be executed in supervisor mode. The routine must be terminated with an RTS instruction. This routine should not be used from BASIC.

## SUB puntaes

A statement that de-allocates the AES from memory. It has absolutely no effect with ROM TOS.

## FUNCTION blitmode%(BYVAL flag%)

This function allows a program to read whether a blitter is connected and being used and turn it on or off. The `flag%` parameter should be -1 to read the configuration, or a new value to set it. The return value is the new configuration, in this binary form:

|                  |   |                                 |
|------------------|---|---------------------------------|
| <i>bit 0</i>     | 0 | use software to blit            |
|                  | 1 | use blitter to blit             |
| <i>bit 1</i>     | 0 | no blitter attached (read-only) |
|                  | 1 | blitter attached (read-only)    |
| <i>bits 2-14</i> |   | reserved                        |
| <i>bit 15</i>    |   | must be zero                    |

This call will work on *all* ROM-based versions of the XBIOS.

## FUNCTION floprate%(BYVAL devno%, BYVAL newrate%)

This function (introduced with TOS 1.04) sets the floppy disk drive step rate. `devno%` gives the device number (0=A,1=B). The `newrate%` parameter gives the rate to set in milli-seconds. Alternatively using a rate of -1 will return the current stepping rate for that device.

## FUNCTION bconmap&(BYVAL devno%)

This function is used for re-mapping the serial ports on the TT and the MegaSTE.

## FUNCTION dmaread&(BYVAL sector&, BYVAL count%, BYVAL buffer&, BYVAL devno%)

This function which was added in TT TOS directly reads `count%` sectors starting at `sector&` from device `devno%` to the buffer at `buffer&`. `devno%` values 0-7 refer to ACSI devices 0-7 and 8-15 refer to SCSI devices 0-7. This function returns a BIOS error code.

## FUNCTION dmawrite&(BYVAL sector&, BYVAL count%, BYVAL buffer&, BYVAL devno%)

This function which was added in TT TOS directly writes `count%` sectors starting at `sector&` from device `devno%` to the buffer at `buffer&`. As a result it should be used with extreme care. `devno%` values 0-7 refer to ACSI devices 0-7 and 8-15 refer to SCSI devices 0-7. This function returns a BIOS error code.

## FUNCTION NVMAccess%(BYVAL opcode%, BYVAL start%, BYVAL count%, BYVAL buffer&)

This function is for accessing the Non Volatile Memory in the TT's real time clock chip. The use of this memory will be dictated by Atari; at the time of writing they had not assigned any bytes so perhaps you should talk to them before using this!

## FUNCTION EsetShift%(BYVAL shiftmode%)

This call is used to set the TT's entire video shift register. The meaning of the bits of `shiftmode%` is as shown below:

| <i>Bit 15</i> | <i>Bit 12</i> | <i>Bits 10-8</i>   | <i>Bits 3-0</i>     |
|---------------|---------------|--|---------------------|
| Smear Mode    | Grey Mode     | Screen mode:<br>000 ST low<br>001 ST medium<br>010 ST high<br>100 TT medium<br>110 TT high<br>111 TT low | Current colour bank |

If you are only interested in the setting of part of the register it is best to use one of the more specific calls below or in the case of the screen mode getrez. `esetshift` was introduced in TT TOS and *requires* the TT video hardware.

## FUNCTION EgetShift%

This call is used to return the current state of the TT's video shift register. The meaning of the bits is as shown above. If you are only interested in the setting of part of the register it is best to use one of the more specific calls below. This call was introduced in TT TOS and *requires* the TT video hardware.

## FUNCTION EsetBank%(BYVAL bank%)

This call sets the TT's palette bank( if the value is 0-15) or just returns it if the value of `bank%` is negative.



The TT's colour lookup table has 256 entries for use in TT low resolution mode. This call lets you select which bank (collection of 16 entries) will be used in the other modes, thus enabling you to switch between palettes very easily. When setting the bank number, the new bank's colours are copied to the old ST colour mode register. This call was introduced in TT TOS and *requires* the TT video hardware.

### FUNCTION EsetColor%(BYVAL colourno%, BYVAL BCDcolour%)

This call is used to read/write a single entry in the TT's colour palette. The colour number should be in the range 0-255 to set the palette or negative to read it. The BCD colour value is encoded as shown below:

bits 15-12      bits 11-8 (Red)      bits 7-4 (Green)      bits 3-0 (Blue)

|        |    |    |    |    |    |    |    |    |    |    |    |    |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|
| Unused | R3 | R2 | R1 | R0 | G3 | G2 | G1 | G0 | B3 | B2 | B1 | B0 |
|--------|----|----|----|----|----|----|----|----|----|----|----|----|

R0 represents the least-significant bit of the red component of the colour, R3 the most-significant. Similarly, G0-G3 give the green component and B0-B3 the blue component.

Note that this (and the other TT specific palette calls) do *not* use the ST compatible method of encoding the colour as per `setcolor` (XBIOS library) and that this call uses the absolute colour number rather than the current bank and so can access all 256 entries. This call was introduced in TT TOS and *requires* the TT video hardware.

### SUB EsetPalette(BYVAL colourno%, BYVAL count%, BYVAL buffer&)

This call is used to set the values of the TT's colour lookup table, or palette. The `buffer&` is a pointer to the area to read the palette from; it would normally be part of a BASIC array (accessed via the `VARPTR` function). `count%` gives the number of colours to change and `colorno%` the first one to modify.

It can be used to set the palette for a single colour, the whole palette or part of it. The colour words are encoded in the standard manner as described under `esetcolor` in this library. This call was introduced in TT TOS and *requires* the TT video hardware.

### SUB EgetPalette%(BYVAL colourno%, BYVAL count%, BYVAL buffer&)

This call is used to read the values of the TT's colour lookup table, or palette. The `buffer&` is a pointer to the area to store the palette from; it would normally be part of a BASIC array (accessed via the `VARPTR` function). `count%` gives the number of colours to read and `colorno%` the first one to use.

It can be used to set the palette for a single colour, the whole palette or part of it. The colour words are encoded in the standard manner as described under `esetcolor` in this library. This call was introduced in TT TOS and *requires* the TT video hardware.

### FUNCTION EsetGray%(BYVAL switch%)

This call is used to read/write the TT video hardware's grey mode bit. Use a `switch%` value of 0 for colour mode, 1 for grey mode or a negative value to read the value. This function returns the old grey mode bit setting.

When grey mode is set, the bottom eight bits of the palette value are used as one of 256 possible grey levels. The best way to see the effect of this is using the Colour part of the Control Panel (just click on the Grey button). This call was introduced in TT TOS and *requires* the TT video hardware.

### FUNCTION EsetSmear%(BYVAL switch%)

This call is used to read/write the TT video hardware's smear mode bit. Use a `switch%` value of 0 for normal, positive for smear mode or a negative value to read the value. This function returns the old smear mode bit setting.

When smear mode is set, the video hardware displays video pixels with value 0 as the last non-zero colour rather than colour zero itself. This can be used to change the colour of a filled-polygon by only changing its outline rather than via a complete re-fill. This call was introduced in TT TOS and *requires* the TT video hardware.

## STESOUND library

This library uses the STE sound hardware which is also available on the TT. Before using this library you should check that the appropriate hardware is present using:

```
IF (GETCOOKIE("_SND") AND 2) = 0 THEN
PRINT "Sorry this program requires DMA SOUND hardware"
STOP -1
```

### SUB DMAcontrol(BYVAL newvalue%)

This sets the STE sound hardware's control register. The appropriate values are:

|   |                     |
|---|---------------------|
| 0 | Disable DMA sound   |
| 1 | Do sound once       |
| 3 | Do sound repeatedly |

SUB DMAsound(BYVAL startadr&, BYVAL endadr&,  
BYVAL mode%)

This sets the start and end addresses of the sample address for the DMA sound hardware together with the sound mode. The mode% gives the sample rate (see the table below) and also whether mono or stereo is to be used. Add 128 in the values in the table for mono.

|   |          |
|---|----------|
| 0 | 6258Hz   |
| 1 | 12517Hz  |
| 2 | 25033Hz  |
| 3 | 50066 Hz |

SUB MWout(BYVAL mask%,BYVAL data%)

Writes the values mask% and data% to the STE MICROWIRE interface. The full generality of this is not used, but it can be used to set the volume/tone and mixing of the STE DMA sound at present. This command issues the command immediately but waits for the command to be complete before returning.

## The GEMVDI library

The GEM VDI library contains routines to access all the VDI routines except the Escape codes. The names of these routines are based on the standard C names for the functions and are listed in the same order as in the Digital Research documentation. This library forces the output to be a GEM program and also ensures that underlines are allowed in variable names.

In the following description, it is assumed that you have used `defint a-z`. Otherwise you must type % at the end of all the function and parameter names that do not have an explicit type.

Most of the parameters are passed by value; those that are not are prefaced with the keyword `varptr`. Naturally the keyword `varptr` is not used when calling the sub-program or function. Generally you should give these a simple variable name to store the returned value. If you are not interested in a particular value just pass 0; the result will then be ignored.

Note that the VDI handle is not passed to the individual routines. This is normally assumed to be the handle used by the run-time system but can be changed used the special `CHANGE_HANDLE` call below. You also do not normally need to open or close a Virtual Workstation; the run-time system does this for you.

We have included functions that we have tested to work on standard STs with the operating system in ROM and also tested additional features available with Atari GDOS version 1.1. Where applicable the following documentation notes when GDOS is required. All these routines use the same pseudo-arrays that the ST BASIC style facilities use.

Note that if you call GEM VDI using these routines and then call the BASIC routines built into the runtimes such as `CLS` and `LINEF`, then strange effects may occur.

If you haven't used the VDI before (or if you have only dabbled) the number of functions can be intimidating. However they are grouped into 'families' and can generally be identified by the first letters of their names. Routines starting with

|    |  |
|----|--|
| v_ | generally do things immediately such as draw circles |
| vs | generally set variables                              |
| vq | return the current settings of variables.            |
| vr | raster routines.                                     |

Most of the GEM VDI routines that draw graphics use a particular set of variables to determine what is drawn, either the line, text, fill area or marker attributes. These are listed below together with the names of the routines which set them:

The *line* attributes:

line type (`vs1_type`), line colour index (`vs1_color`), line width (`vs1_width`) and end style (`vs1_ends`).

Note that all these routines start with `vs1_`.

The *text* attributes:

text alignment (`vst_alignment`), colour (`vst_color`), height (`vst_height` and `vst_point`), rotation (`vst_rotation`), font (`vst_font`) and special effects (`vst_effects`).

Note that all these routines start with `vst_`.

The *fill* area attributes:

fill area colour (`vsf_color`), fill style (`vsf_interior`), fill style index (`vsf_style`) and perimeter visibility (`vsf_perimeter`).

The *marker* attributes:

marker colour (`vsm_color`), marker height (`vsm_height`) and marker type (`vsm_type`). Note that all these routines start with `vsm_`.

Angles are measured from 0 to 3600 anti-clockwise with zero being the positive x axis. VDI rectangles are specified using the top left and bottom right corners.

## GEMVDI : Control

These functions can be ignored if you are new to using the VDI.

### FUNCTION GDOS

returns true (-1) if the program is running under GDOS. This means that the program can load fonts and open workstations. GDOS returns 0 if GDOS is missing. If you are writing a program to run under GDOS check that it is there at the beginning of your program; otherwise the machine may crash when you load fonts.

### FUNCTION CVHANDLE

returns the current handle being used by this library. Should be saved before opening a physical or virtual workstation.

### SUB CHANGE\_HANDLE(BYVAL handle)

modifies the handle used by this library to be handle.

### FUNCTION RESOLUTION

although not really a VDI function it can be useful when opening a GDOS workstation returns the current screen resolution according to the following table:

This is obtained via the XBIOS getrez call.

### SUB v\_opnwk(BYVAL inv(), handle, BYVAL outv())

This routine attempts to open a workstation. Do not use it unless you have GDOS installed; otherwise the machine will crash. The workstation handle allocated is returned in handle or is 0 if the device cannot be opened. The handle returned also becomes the current handle for the other routines in this library to use.

When using this routine you must ensure that there is sufficient system memory to load the driver using the Keep and Leave options. inv() specifies the device to open and various initial values for variables as follows on the next page:

|        |  |
|--------|--|
| inv(0) | the device id number as used in the ASSIGN.SYS file. |
| inv(1) | line type  |
| inv(2) | line colour  |
| inv(3) | marker type  |
| inv(4) | marker colour  |
| inv(5) | text font  |
| inv(6) | text colour  |
| inv(7) | fill area interior                                   |
| inv(8) | fill area style index                                |
| inv(9) | fill area colour                                     |

inv(10) NDC to RC transformation flag; 2 use device (pixel) co-ordinates; 0 use normalised co-ordinates 0-32767.

The outv() array gives information concerning the facilities available on the device. It must have at least 57 elements.

### SUB v\_clswk

closes the current workstation. You must call this for each workstation that you open. This causes the page to output to printer devices. You should follow this with a CHANGE\_HANDLE call. Do not attempt to close the run-time system's workstation with this call.

### SUB v\_opnwk(BYVAL inv(), handle, BYVAL outv())

This attempts to open a new virtual workstation for the device with handle given in handle and returns the new handle in handle. The parameters are the same as the v\_opnwk call with the exception of handle which must be set up before calling this routine. If using GDOS the handle to pass is returned by the v\_opnwk call above. When not using GDOS, v\_opnwk can also be used to have different clipping rectangles, fill patterns etc on the screen at once without the need to change all the variables.

In this case make sure that handle is 1 before calling this routine. Zero is returned in handle if the device cannot be opened. The handle returned becomes the new handle to be used by the rest of this library. If you are not using GDOS you can *not* use NDC co-ordinates; you *must* give inv(10) as 2.

### SUB v\_clsvwk

closes the current virtual workstation. You must call this for each virtual workstation that you open yourself. You should follow this with a CHANGE\_HANDLE call.

### SUB v\_updwk

This routine updates the current workstation. Not applicable to screens, but will cause a printer or plotter to output the current page.

### SUB v\_clrwk

This routine clears the current workstation (usually the screen).

## FUNCTION vst\_load\_fonts

Loads fonts into GEMDOS free memory and returns the number of fonts loaded as the result. This call only works when GDOS is installed; otherwise the machine may crash. The amount of GEMDOS free memory may be increased using the compiler's Keep and Leave options. See *Appendix B* and *Chapter 3* of the *User Manual*. These options let you leave more memory to load fonts.

## SUB vst\_unload\_fonts

Unloads fonts that were loaded using `vst_load_fonts`. This *must* be called if you have used `vst_load_fonts` before the Virtual Workstation is closed. The default Virtual Workstation is closed by the run-time system when a program terminates.

## SUB vs\_clip(BYVAL flag, BYVAL x1, BYVAL y1, BYVAL x2, BYVAL y2)

If `flag=0` then switches VDI clipping off; otherwise sets the clip rectangle to be (x1,y1) to (x2,y2).

# GEMVDI : Drawing Primitives

## SUB v\_pline(BYVAL n, BYVAL pts())

Draws a series of lines between `n` points. The points are passed in the array `pts()` with `pts(0)` containing the first x co-ordinate; `pts(1)` the first y co-ordinate; `pts(2)` the second x co-ordinate; `pts(3)` the second y co-ordinate etc. `n` must be at least two. The *line* attributes are used.

## SUB v\_pmarker(BYVAL n, BYVAL pts())

Draws a series of markers at `n` points. The points are passed in the array `pts()` with `pts(0)` containing the first x co-ordinate; `pts(1)` the first y co-ordinate; `pts(2)` the second x co-ordinate; `pts(3)` the second y co-ordinate etc. `n` must be at least one. The *marker* attributes are used.

## SUB v\_gtext(BYVAL x, BYVAL y, BYVAL text\$)

Writes the graphics text `text$` to the display at (x,y). The *text* attributes are used.

## SUB v\_fillarea(BYVAL n, BYVAL pts())

Fills a polygon given `n` points. The points are passed in the array `pts()` with `pts(0)` containing the first x co-ordinate; `pts(1)` the first y co-ordinate; `pts(2)` the second x co-ordinate; `pts(3)` the second y co-ordinate etc. `n` must be at least two. The first point is connected to the last point; there is no need to specify the first point twice. The *fill area* attributes are used.

## SUB v\_contourfill(BYVAL x, BYVAL y, BYVAL colour)

Seed fills an area starting with (x,y) in the colour given by `colour`. If `colour` is negative the algorithm searches for any colour other than that specified. The *fill area* attributes are used.

## SUB vr\_rectfl(BYVAL x1, BYVAL y1, BYVAL x2, BYVAL y2)

fills the rectangle given by the points (x1,y1) and (x2,y2). Uses the *fill area* attributes except that no box is drawn around it whatever the value given to `vsf_perimeter`.

## SUB v\_bar x1, y1, x2, y2

fills the rectangle using the *fill area* attributes and surrounds it with a box in the current *fill area* colour if `vsf_perimeter` has been set to 1.

## SUB v\_arc(BYVAL x, BYVAL y, BYVAL r, BYVAL start\_angle, BYVAL end\_angle)

draws an arc using the *line* attributes centred on (x,y) of radius `r` starting with angle `start_angle` and ending with angle `end_angle`.

## SUB v\_pieslice(BYVAL x, BYVAL y, BYVAL r, BYVAL start\_angle, BYVAL end\_angle)

draws a pie slice using the *fill area* attributes centred on (x,y) of radius `r` starting with angle `start_angle` and ending with angle `end_angle`.

## SUB v\_circle(BYVAL x, BYVAL y, BYVAL r)

draws a circle using the *line* attributes centred on (x,y) of radius `r`.



SUB v\_ellarc(BYVAL x,BYVAL y,BYVAL xr,BYVAL  
BYVAL yr, BYVAL start\_angle, BYVAL  
end\_angle)

draws an elliptical arc using the *line* attributes centred on (x,y) with x radius xr and y radius yr starting with angle start\_angle and ending with angle end\_angle.

SUB v\_ellpie(BYVAL x, BYVAL y, BYVAL xr, BYVAL  
yr, BYVAL start\_angle, BYVAL end\_angle)

draws an elliptical pie slice using the *fill area* attributes centred on (x,y) with x radius xr and y radius yr starting with angle start\_angle and ending with angle end\_angle.

SUB v\_ellipse(BYVAL x, BYVAL y, BYVAL xr,  
BYVAL yr)

draws an ellipse using the *line* attributes centred on (x,y) with x radius xr and y radius yr.

SUB v\_rbox(BYVAL x1, BYVAL y1, BYVAL x2, BYVAL  
y2)

draws a box with rounded edges using the *line* attributes with co-ordinates (x1,y1) to (x2,y2).

SUB v\_rfbox(BYVAL x1, BYVAL y1, BYVAL x2,  
BYVAL y2)

draws a filled rectangle with rounded edges using the *fill area* attributes with co-ordinates (x1,y1) to (x2,y2).

SUB v\_justified(BYVAL x, BYVAL y, test\$, BYVAL  
length, BYVAL wordflag, BYVAL charflag)

draws the justified text test\$ at (x,y) in pixel width of length. If wordflag and charflag are non-zero then GEM VDI may adjust the word and character spacing. So to only adjust the word (not the character) spacing use wordflag=1 and charflag=0. The text attributes are used.

## GEMVDI: Attributes

SUB vswr\_mode(BYVAL mode)

Sets the current writing mode that is used in all the drawing routines to be mode. The different modes are

| Number | Mode                |
|--------|---------------------|
| 1      | Replace             |
| 2      | Transparent         |
| 3      | XOR                 |
| 4      | Reverse transparent |

SUB vs\_color(BYVAL index, BYVAL r, BYVAL g,  
BYVAL b)

Sets the colour numbered index to be the nearest possible to red, green and blue intensities given by r, g and b. Intensities vary from 0 to 1000. The default colour indices depend on whether you have used the Control Panel or the PALETTE statement. If you have modified the colours using the Control Panel you will obviously get the colours that you have set. However simply running the Control Panel also has the effect of changing the colours that are used in Low resolution. The PALETTE statement also sets these colours:

| Number | With Control Panel | Without       |
|--------|--------------------|---------------|
| 0      | White              | White         |
| 1      | Black              | Black         |
| 2      | Red                | Red           |
| 3      | Green              | Green         |
| 4      | Blue               | Blue          |
| 5      | Dark blue          | Cyan          |
| 6      | Brown              | Yellow        |
| 7      | Dark Green         | Magenta       |
| 8      | Grey               | Light Grey    |
| 9      | Dark Grey          | Dark Grey     |
| 10     | Light Blue         | Light Red     |
| 11     | Blue Green         | Light Green   |
| 12     | Light Purple       | Light Blue    |
| 13     | Dark Purple        | Light Cyan    |
| 14     | Dark Yellow        | Light Yellow  |
| 15     | Light Yellow       | Light Magenta |

### SUB vs1\_type(BYVAL type)

sets the line type to be one of

| type | action  |
|------|---|
| 1    | solid   |
| 2    | long dash                                       |
| 3    | dot   |
| 4    | dash dot  |
| 5    | dash  |
| 6    | dash, dot, dot                                  |
| 7    | user defined style set by vs1_udsty (see below) |

This is one of the *line* attributes.

### SUB vs1\_udsty(BYVAL pattern)

sets the user defined line style (see vs1\_type above) to be given by the integer parameter. The bits from this word are repeated when drawing the line so that &hFFFF (all bits set) corresponds to a solid line, &hFFF0 (first 12 bits set last 4 reset) corresponds to long dashes etc.

### SUB vs1\_width(BYVAL width)

sets the line width to width pixels. width should be odd; if an even number is given then one less than this is used. This is one of the *line* attributes.

### SUB vs1\_color(BYVAL colour)

sets the current line colour index to be the colour index specified. If this is too large then colour index one is used. This is one of the *line* attributes. For the default values for the colour indices and how to change them see vs\_color above.

### SUB vs1\_ends(BYVAL beginstyle, BYVAL endstyle)

Sets the styles for the beginning and end of lines to be beginstyle and endstyle respectively. The values for styles are as follows:

|   |                  |
|---|------------------|
| 0 | square (default) |
| 1 | arrow            |
| 2 | rounded          |

This is one of the *line* attributes.

### SUB vsm\_type(BYVAL type)

sets the marker type to be one of

| Number | Type           |
|--------|----------------|
| 1      | dot            |
| 2      | plus           |
| 3      | asterisk       |
| 4      | square         |
| 5      | Diagonal cross |
| 6      | Diamond        |

This is a *marker* attribute.

### SUB vsm\_height(BYVAL height)

Attempts to set the marker height to height. If the given value is not possible then the next smallest height is set and returned with the x value in ptsout and the y value in ptsout+2. This is a *marker* attribute.

### SUB vsm\_color(BYVAL colour)

sets the current marker colour index to be the colour index specified. If this is too large then colour index 1 is used. This is one of the *marker* attributes. For the default values for the colour indices and how to change them see vs\_color above.

### SUB vst\_height(BYVAL height)

Attempts to set the text height to height. If the given value is not possible then the next smallest height is set. Returned in ptsout is the width of the widest character in the font. ptsout+2 contains the height of the highest character in the font. ptsout+4 and ptsout+6 contain the dimensions of the cell in which such characters are drawn. This is a *text* attribute function.

### SUB vst\_point(BYVAL points)

Attempts to set the text height in points to points . If the given value is not possible then the next smallest point size is set and returned in intout. Returned in ptsout is the width of the widest character in the font in pixels. ptsout+2 contains the height of the highest character in the font in pixels. ptsout+4 and ptsout+6 contain the dimensions of the cell in which such characters are drawn. This is a *text* attribute function.

### SUB vst\_rotation(BYVAL angle)

Attempts to set the orientation of the text that is drawn. angle should be one of

| angle | effect        |
|-------|---------------|
| 0     | Normal        |
| 900   | Up the page   |
| 1800  | Upside down   |
| 2700  | Down the page |

Don't pass an angle greater than 3150; bus errors may result. This is a *text* attribute function.

### SUB vst\_font(BYVAL font)

Sets the current font. The standard system font is font 1. To load other fonts you need GDOS. To find the names and indices of font use vqt\_name. This is a *text* attribute function.

### SUB vst\_color(BYVAL colour)

sets the current text colour index to be the colour index specified. If this is too large then colour index one is used. This is one of the *text* attributes. For the default values for the colour indices and how to change them see vs\_color above.

### SUB vst\_effects(BYVAL effects)

Sets the special effects that are used when drawing text. effects is a bit map with masks if set having effects as follows

| Mask | Action               |
|------|----------------------|
| 1    | Thickened            |
| 2    | Grey (not bright)    |
| 4    | Skewed (like italic) |
| 8    | Underlined           |
| 16   | Outline              |
| 32   | Shadowed             |

This is a *text* attribute function.

### SUB vst\_alignment(BYVAL horizontal, BYVAL vertical)

Sets the alignment that is used when text is drawn. This is where the given co-ordinate refers to when using v\_gtext. horizontal may have the following values:

| value | effect                   |
|-------|--------------------------|
| 0     | left justified (default) |
| 1     | centre justified         |
| 2     | right justified          |

vertical may have the following values:

| value | effect   |
|-------|--|
| 0     | base line (default). The bottom of characters without descenders.  |
| 1     | half line. The top of lower case letters such as a and e.          |
| 2     | ascent line. The top of upper case letters such as A and E.        |
| 3     | bottom. The very bottom of the character cell.                     |
| 4     | descent. The bottom of characters with descenders such as g and y. |
| 5     | top. The very top of the character cell.                           |

This is a *text* attribute function.

### SUB vsf\_interior(BYVAL fill\_interior)

This sets the fill style to be one of the following according to fill\_interior:

| Style | Effect             |
|-------|--------------------|
| 0     | hollow             |
| 1     | solid              |
| 2     | pattern            |
| 3     | hatch              |
| 4     | user-defined style |

For values 2 and 3 this is used in conjunction with the style index set by vsf\_style. The user-defined style is set using vsf\_updat and is an Atari logo by default. This is a *fill area* function and can also be set using the COLOR statement.

### SUB vsf\_style(BYVAL style\_index)

Sets the current style\_index to be used in conjunction with the pattern and hatch fill interiors. This is a *fill area* function and can also be set using the COLOR statement.

See the COLOR statement in *Command Reference* for the fill styles.

### SUB vsf\_color(BYVAL colour)

sets the current fill area colour index to be the colour index specified. If this is too large then colour index one is used. This is one of the *fill area* attributes. For the default values for the colour indices and how to change them see vs\_color above.

### SUB vsf\_perimeter(BYVAL flag)

If flag is non-zero then filled areas are surround by a perimeter using the current line attributes. This is the default. If flag=0 then no perimeter is drawn. This is a *fill area* function.

SUB vsf\_updat(BYVAL plane(), BYVAL planes)

Sets the user definable fill pattern that is selected using vsf\_interior 4. planes gives the number of planes in the fill pattern (for monochrome this is one). Any unspecified planes are zeroed when filling takes place. The pattern is passed in the array plane() with 16 words for each plane. The first word (element zero in the array) gives the bit pattern for the top line and the last one the bottom line. Within each word the most significant bit gives the left most pixel. This is a *fill area* function.

## GEMVDI: Raster Functions

SUB vro\_cpyfm(BYVAL mode, BYVAL xy(), BYVAL source&, BYVAL dest&)

This performs an opaque raster copy with the source rectangle specified in xy(0) to xy(3) and the destination rectangle in xy(4) to xy(7). source& is the source Memory Form Definition Block (MFDB) and dest& is the destination MFDB. Note that the device dependent form appears always to be used by current versions of the operating system, regardless of the indicated mode. mode specifies the operation performed. In the table below S is the source value and D the destination value.

| mode | new value     |         |
|------|---------------|---------|
| 0    | 0             |         |
| 1    | S AND D       |         |
| 2    | S AND (NOT D) |         |
| 3    | S             | replace |
| 4    | (NOT S) AND D | erase   |
| 5    | D             |         |
| 6    | S XOR D       | XOR     |
| 7    | S OR D        |         |
| 8    | NOT (S OR D)  |         |
| 9    | NOT (S XOR D) |         |
| 10   | NOT D         |         |
| 11   | S OR (NOT D)  |         |
| 12   | NOT S         |         |
| 13   | (NOT S) OR D  |         |
| 14   | NOT (S AND D) |         |
| 15   | 1             |         |

The main differences between the built-in GET and PUT statements and the facilities available via this function are that no clipping is performed and you can move directly on screen without going via an array.

SUB vrt\_cpyfm(BYVAL mode, BYVAL xy(), BYVAL source&, BYVAL dest&, BYVAL foreground, BYVAL background)

This performs a raster copy from a monochrome raster block with the source rectangle specified in xy(0) to xy(3) to a colour block with destination rectangle in xy(4) to xy(7). source& is the source Memory Form Definition Block (MFDB) and dest& is the destination MFDB. mode specifies the operation performed and is given by the following table:

| Number | Mode                |
|--------|---------------------|
| 1      | Replace             |
| 2      | Transparent         |
| 3      | XOR                 |
| 4      | Reverse transparent |

foreground and background specify the corresponding colours to be used in the destination block.

SUB v\_get\_pixel(BYVAL x, BYVAL y, value, index)

Gives the value of the pixel and (x,y) in value and the colour index corresponding to this in index. Normally index and value are the same.

## GEMVDI: Mouse & Keyboard State Functions

SUB v\_show\_c(BYVAL flag)

Shows the graphics cursor on the screen; unless flag<>0 and v\_hide\_c has been called more than once previously. If flag<>0 and v\_hide\_c has been called *n* times then you must call v\_show\_c *n* times before the mouse is displayed again. If flag=0 the mouse is displayed regardless of the number of calls to v\_hide\_c.

SUB v\_hide\_c

Removes the mouse from the screen.

### SUB vsc\_form(BYVAL inv())

Sets the current shape of the mouse. The elements of the array `inv()` are as follows:

|                                |                              |
|--------------------------------|------------------------------|
| <code>inv(0)</code>            | x co-ordinate of hot spot.   |
| <code>inv(1)</code>            | y co-ordinate of hot spot    |
| <code>inv(2)</code>            | reserved must be 1           |
| <code>inv(3)</code>            | mask colour index normally 0 |
| <code>inv(4)</code>            | data colour index normally 1 |
| <code>inv(5) - inv(20)</code>  | bitmap for mouse mask        |
| <code>inv(21) - inv(36)</code> | bitmap of mouse data         |

### SUB vq\_mouse(button, x, y)

Returns the current position of the mouse in (x,y) and which mouse buttons are down in `button`. Bit 0 of `button` is 1 if the left button is down and bit 1 is down if the right button is down.

### SUB vq\_key\_s( kstate)

returns the current state of the Shift keys in `kstate` as a bit map with the bit set if the corresponding key is down:

| Bit | Mask | Meaning     |
|-----|------|-------------|
| 0   | 1    | Right Shift |
| 1   | 2    | Left Shift  |
| 2   | 4    | Control     |
| 3   | 8    | Alt Key     |

## GEMVDI: Enquires

### SUB vq\_extnd(BYVAL flag, BYVAL info())

Returns information about the current VDI variables. If `flag=0` then the Open Workstation values are returned and if `flag=1` the Extended Inquire values are returned. `info()` must have at least 57 elements. The values returned are too numerous to list here. `info(0)` to `info(44)` contain the values returned in `intout` and `info(45)` to `info(56)` contain the values in `ptsout`.

### SUB vq\_color(BYVAL colour\_index, BYVAL flag, BYVAL rgb())

returns the red, green and blue intensities for `colour_index` in `rgb(0)`, `rgb(1)` and `rgb(2)` respectively. If `flag=0` the values that the device gives are returned. If `flag=1` then the values that the user last requested are given.

### SUB vql\_attributes(BYVAL info())

returns the current *line* attributes as follows

|                      |   |
|----------------------|---|
| <code>info(0)</code> | line type (set with <code>vsl_type</code> )             |
| <code>info(1)</code> | line colour (set with <code>vsl_color</code> )          |
| <code>info(2)</code> | writing mode (set with <code>vswr_mode</code> )         |
| <code>info(3)</code> | start point end style (set with <code>vsl_ends</code> ) |
| <code>info(4)</code> | end point end style (set with <code>vsl_ends</code> )   |
| <code>info(5)</code> | line width (set with <code>vsl_width</code> )           |

### SUB vqm\_attributes(BYVAL info())

returns the current *marker* attributes as follows

|                      |   |
|----------------------|---|
| <code>info(0)</code> | marker type (set with <code>vsm_type</code> )     |
| <code>info(1)</code> | marker colour (set with <code>vsm_color</code> )  |
| <code>info(2)</code> | writing mode (set with <code>vswr_mode</code> )   |
| <code>info(3)</code> | not used  |
| <code>info(4)</code> | marker height (set with <code>vsm_height</code> ) |

### SUB vqf\_attributes(BYVAL info())

returns the current *fill area* attributes as follows

|                      |  |
|----------------------|--|
| <code>info(0)</code> | fill area interior style (set with <code>vsf_interior</code> ) |
| <code>info(1)</code> | fill area colour (set with <code>vsf_color</code> )            |
| <code>info(2)</code> | fill area style (set with <code>vsf_style</code> )             |
| <code>info(3)</code> | writing mode (set with <code>vswr_mode</code> )                |
| <code>info(4)</code> | fill perimeter status (set with <code>vsf_perimeter</code> )   |

### SUB vqt\_attributes(BYVAL info())

returns the current *text* attributes as follows:

|                      |   |
|----------------------|---|
| <code>info(0)</code> | current font (set with <code>vst_font</code> )  |
| <code>info(1)</code> | text colour (set with <code>vst_color</code> )  |
| <code>info(2)</code> | text rotation base line (set with <code>vst_rotation</code> )                             |
| <code>info(3)</code> | horizontal alignment (set with <code>vst_alignment</code> )                               |
| <code>info(4)</code> | vertical alignment (set with <code>vst_alignment</code> )                                 |
| <code>info(5)</code> | writing mode (set with <code>vswr_mode</code> )   |
| <code>info(6)</code> | max. char. width (set implicitly via <code>vst_height</code> or <code>vst_point</code> )  |
| <code>info(7)</code> | max. char. height (set implicitly via <code>vst_height</code> or <code>vst_point</code> ) |
| <code>info(8)</code> | max. cell width (set implicitly via <code>vst_height</code> or <code>vst_point</code> )   |
| <code>info(9)</code> | max. cell height (set implicitly via <code>vst_height</code> or <code>vst_point</code> )  |



SUB vqt\_extent(BYVAL text\$, BYVAL info())

Returns in info() the co-ordinates of a box needed to enclose the text string text\$ if written using the current text attributes. This is used when writing text using different heights, fonts, effects etc. The co-ordinates returned are

|                   |              |
|-------------------|--------------|
| info(0) & info(1) | bottom left  |
| info(2) & info(3) | bottom right |
| info(4) & info(5) | top right    |
| info(6) & info(7) | top left     |

SUB vqt\_width(BYVAL char, cellwidth, left, right)

Returns the width of the cell that the character char uses with the current text attributes in cellwidth; together with number of blank pixels the space to the left in left and the number of blank pixels to the right in right.

FUNCTION vqt\_name(BYVAL number, name\$)

Returns as the result of the function the font index used in vst\_font from number in the range 1 to the number of fonts. Also returned is the name of the font in name\$.

SUB vqt\_fontinfo(firstchar, lastchar, BYVAL distances(), maxwidth, BYVAL effects())

Returns information on the current font. firstchar and lastchar give the range of characters available. distances() returns information on the size of the characters as the distances between the following lines and the base line (the bottom of characters without descenders).

|              |                                      |
|--------------|--------------------------------------|
| distances(0) | very bottom of the cell descenders   |
| distances(1) | bottom of characters with descenders |
| distances(2) | the top of normal lower case letters |
| distances(3) | the top of upper case letters        |
| distances(4) | the top of the cell                  |

maxwidth is the maximum cell width not including any special effects.

|            |   |
|------------|---|
| effects()  | contains information on the effect of the current special effects set using vst_effects as follows: |
| effects(0) | additional pixels in the x direction  |
| effects(1) | the number of pixels that the left hand of the character cell is slanted at the baseline.           |
| effects(2) | the number of pixels that the top right is slanted relative to the base line.                       |

## The GEMAES Library

The GEMAES library lets you use the user-interface facilities of GEM such as menus and dialog boxes. As with the VDI there are a large number of routines, but these are divided into families. Unfortunately many of the routines use data structures which are too numerous to describe here. See the books on GEM described in the *Bibliography* in the *User Manual*.

The names of these routines are based on the standard C names for the functions and are listed in the same order as in the Digital Research documentation. This library forces the output to be a GEM program and also ensures that underlines are allowed in variable names.

As with the GEM VDI library, in the following descriptions, it is assumed that you have used `defint a-z`. Otherwise you must type % at the end of all the function and parameter names that do not have an explicit type.

Most of the parameters are passed by value; those that are not are prefaced with the keyword `varptr`. This does not mean that the keyword `varptr` is specified when calling the sub-program or function. Generally you should give these a simple variable name to store their values. If you are not interested in a particular value just pass 0; the result will then be ignored.

To help you use the GEMAES data structures there is a file called GEMAES.BH on your master disk which contains most of the constant definitions you need. This takes a few seconds to compile, but saves you the trouble of looking up their values. Hopefully it will also encourage you to make your programs more readable by using these constants rather than the actual numbers.

One way in which the AES differs from the VDI is that rectangles are passed as x, y, w and h. (x, y) gives the top left corner; w the width and h the height. The AES uses window handles which are not the same as VDI handles.

Please see the next chapter and the example programs on the disks for examples of the use of these functions.

When using the HiSoft BASIC WINDOW commands the run-time system will 'steal' any update messages that would normally be generated, so if you are calling the GEM AES window handling routines directly then you should use WINDOW OFF to stop this.

To remove the window that the run-time system opens when a program starts running use WINDOW CLOSE, or to stop it opening at all use the Window Defeat option, described in *Appendix B* and *Chapter 3* of the *User Manual*.

You may use resource files with HiSoft BASIC; please see the `rsrc_load` call and the next chapter for more details.

## GEMAES: Utility routine

SUB INTERSECTION(BYVAL x1, BYVAL y1, BYVAL w1, BYVAL h1, BYVAL x2, BYVAL y2, BYVAL w2, BYVAL h2, x, y, w, h)

When using the AES windows in conjunction with desk accessories you often need to find the intersection of two rectangles so you know what you need to update. This routine returns in (x, y, w, h) the intersection of the two rectangles (x1, y1, w1, h1) and (x2, y2, w2, h2).

## GEMAES: Message Passing

SUB appl\_read(BYVAL ap\_id, BYVAL length, BYVAL message&)

Reads length bytes from the message pipe of the application with identifier ap\_id into the buffer pointed to by message&. The ap\_id is normally that of the program itself obtained from the global array pointed to by GB.

SUB appl\_write(BYVAL ap\_id, BYVAL length, BYVAL message&)

Writes length bytes to the message pipe of the application with identifier ap\_id from the buffer pointed to by message&. The ap\_id is normally that of a desk accessory obtained using the appl\_find call below. To use this effectively you need to know full technical details of the desk accessory.

FUNCTION appl\_find(BYVAL name\$)

returns the identifier of the application called name\$ for use with the appl\_write statement. There is no need to ensure that name\$ is 8 characters long and padded with spaces; the library routine will do this for you.

## GEMAES: Events

FUNCTION evnt\_keybd

returns the next key pressed in a unique form. The low byte is the ASCII value and the high byte is the scan code for the key unless CTRL is held down together with cursor left, cursor right or clr/home keys when &h73, &h74 and &h77 are returned in the high byte respectively.

FUNCTION evnt\_button(BYVAL clicks, BYVAL mask, BYVAL state, xout, yout, button, kstate)

This function waits for a particular mouse button state. clicks is the maximum number of clicks that the application is waiting for. The number of clicks that actually occurred (between 1 and clicks inclusive) is returned as the result of the function. mask, state and button use a bit mask for the state of the button. Bit 0 is the left button and bit 1 is the right button. A value of 0 means the button is up and 1 means the button is down. The button state is ANDed with mask and tested to be equal to state. The state of all the buttons is returned in button. x and y give the position of the mouse when the event occurred and kstate gives the state of the Shift keys. With the following bits giving the state:

| Mask | Key         |
|------|-------------|
| 1    | right Shift |
| 2    | left Shift  |
| 4    | Ctrl        |
| 8    | Alt         |

As usual a bit is 0 if the key is up and 1 if the key is down.

SUB evnt\_mouse(BYVAL flag, BYVAL x, BYVAL y, BYVAL w, BYVAL h, xout, yout, button, kstate)

Waits for the mouse to enter or leave a rectangle given by (x, y) with width w and height h. If flag=0 then the routine returns on entry else if flag=1 it returns on exit. The position of the mouse when the event occurs is given in (xout, yout) and the state of the mouse buttons and Shift keys are given in button and kstate as defined in evnt\_button immediately above.

SUB evnt\_mesag(BYVAL message&)

Waits for a message in the applications message pipe and places this in the buffer pointed to by message&. This must be at least 16 bytes long.

SUB evnt\_timer(BYVAL time&)

Waits for time& milliseconds to pass.

FUNCTION evnt\_multi(BYVAL eflags, BYVAL clicks, BYVAL mask, BYVAL state, BYVAL flag1, BYVAL x1, BYVAL y1, BYVAL w1, BYVAL h1, BYVAL flag2, BYVAL x2, BYVAL y2, BYVAL w2, BYVAL h2, BYVAL message&, BYVAL time&, xout, yout, button, kstate, key\_pressed, gotclicks)

Waits for a combination of events to occur. The events are given as a mask in eflags with the following mask values:

| Mask | Name      |
|------|-----------|
| 1    | MU_KEYBD  |
| 2    | MU_BUTTON |
| 4    | MU_M1     |
| 8    | MU_M2     |
| 16   | MU_MESAG  |
| 32   | MU_TIMER  |

These constants are defined in the GEMAES.BH file. The parameter key\_pressed is the key pressed as returned by evnt\_keybd. clicks, mask, state, xout, yout, button and kstate are defined under evnt\_button. The value that would be returned by evnt\_button is returned in gotclicks. message& is defined under evnt\_mesag and time& under evnt\_timer. flag1, x1, y1, w1, h1 and the corresponding parameters with 2 refer to two distinct rectangles as defined under evnt\_mouse. The value returned is a mask of the event(s) that actually occurred as specified in the table above. Note that more than one event may be returned at once.

FUNCTION evnt\_dclick(BYVAL newvalue, BYVAL setorget)

This routine sets or gets the double click speed of the mouse. If setorget=1 then the speed is set according to newvalue and it is returned as the result of the function if setorget=0. The values used are between 0 and 4 and correspond to the values on the Control Panel.

## GEMAES: Menus

SUB menu\_bar(BYVAL tree&, BYVAL flag)

Displays a menu bar represented by the object tree pointed to by tree& if flag=1 and "removes" the menu bar if flag=0. In general it is easier to use the MENU function (see above) to display menus. tree& must point to a valid menu tree as produced by a resource construction set or MENU call. If you use this sub-program to display a menu then you should remove it before your program finishes. Note that removing an item does not cause the menu bar to be cleared.

SUB menu\_icheck(BYVAL tree&, BYVAL item, BYVAL flag)

Displays a tick (check mark) by the menu item in the menu object tree tree& if flag=1. If flag=0 then the tick is erased. If you have used the MENU function to display the menu bar then the tree& is the value returned by the function call. The item number is calculated as described under the MENU function.

SUB menu\_ienable(BYVAL tree&, BYVAL item, BYVAL flag)

Dims a menu item in the menu object tree tree& if flag=0. If flag=1 then the item is displayed in full brightness. If you have used the MENU function to display the menu bar then the tree& is the value returned by the function call. The item number is calculated as described under the MENU function.

SUB menu\_tnormal(BYVAL tree&, BYVAL title, BYVAL flag)

Displays a menu title in reverse video from the menu object tree tree& if flag=0. If flag=1 then the item is displayed in normal video. If you have used the MENU function to display the menu bar then the tree& is the value returned by the function call. The title number is calculated as described under the MENU function and is also returned in the message packet following a MN\_SELECTED event.



SUB menu\_text(BYVAL tree&, BYVAL item, BYVAL text\$)

Modifies the menu item to be text\$ in the menu object tree tree& If you have used the MENU function to display the menu bar then the tree& is the value returned by the function call. The item number is calculated as described under the MENU function. text\$ must not be larger than the original text string in the object tree when it was created (e.g. with MENU). If you need to make the string larger call the MENU function again.

## GEMAES: Objects

FUNCTION objc\_add(BYVAL tree&, BYVAL parent, BYVAL child)

Adds an object with index child as a child of the object parent in the object tree tree&. Returns 0 if there is an error or non-zero if no error occurs.

FUNCTION objc\_delete(BYVAL tree&, BYVAL object)

Delete the object from the object tree tree&. This removes the link from the parent to the object. Returns 0 if there is an error or non-zero if no error occurs.

FUNCTION objc\_draw(BYVAL tree&, BYVAL object, BYVAL depth, BYVAL x, BYVAL y, BYVAL w, BYVAL h)

Draws part or all of the object tree tree& starting with object and drawing to a depth of depth using a clipping rectangle starting at (x,y) of width w and height h. If depth=0 then only the starting object is drawn; if depth=1 then the first generation children of the object are drawn and so on. Returns 0 if there is an error or non-zero if no error occurs.

FUNCTION objc\_find(BYVAL tree&, BYVAL object, BYVAL depth, BYVAL x, BYVAL y)

Returns the object number of the item in the object tree tree& which contains the point (x,y). The search starts at object and is to a depth of depth as specified in objc\_draw. If the point is not under any object then -1 is returned.

FUNCTION objc\_offset(BYVAL tree&, BYVAL object, x, y)

Returns in (x,y) the co-ordinates of object from the tree tree& relative to the screen. Returns 0 if there is an error or non-zero if no error occurs.

FUNCTION objc\_order(BYVAL tree&, BYVAL object, BYVAL newpos)

Moves object in the tree tree& to a different position in its parents list of children. If newpos is

- |    |                                |
|----|--------------------------------|
| -1 | moved to the front of the list |
| 0  | moved to the end of the list   |
| 1  | moved to 1 from the end        |
| 2  | moved to 2 from the end etc.   |

This function returns 0 if there is an error or non-zero if no error occurs.

FUNCTION objc\_edit(BYVAL tree&, BYVAL object, BYVAL char, BYVAL index, BYVAL kind, newindex)

Performs the low level editing of the object in tree tree&. The object must be of type G\_TEXT or G\_BOXTEXT. The operation performed depends on the value of kind:

- |   |         |   |
|---|---------|---|
| 1 | ED_INIT | combine the strings from the te_ptext and te_ptmplt fields of the TEDINFO structure into a formatted string and display it with the text cursor on.   |
| 2 | ED_CHAR | validate the character char, which was probably typed by the user, against te_pvalid and update te_ptext and re-display the string. The original index in the string to use is index and the updated value is stored in newindex. |
| 3 | ED_END  | Removes the text cursor.  |

The function returns 0 if there is no error or non-zero if an error occurs.

FUNCTION objc\_change(BYVAL tree&, BYVAL object, BYVAL x, BYVAL y, BYVAL w, BYVAL h, BYVAL newstate, BYVAL flag)

Modifies the ob\_state field of the object in tree tree& to be newstate and if flag=1 redraws the object with a clip rectangle at (x,y) with width w and height h. If flag=0 then the ob\_state is modified but the object is not re-drawn.

The function returns 0 if there is no error or non-zero if an error occurs.

## GEMAES: Forms

### FUNCTION form\_do(BYVAL tree&, BYVAL object)

lets the user fill in a form given by the object tree `tree&` with the interaction starting with object number `object`. `object` must be an editable text field or 0 if the form does not contain any editable text fields. The value returned is the index of the object that caused the exit (e.g. an OK button).

### SUB form\_dial(BYVAL flag, BYVAL x1, BYVAL y1, BYVAL w1, BYVAL h1, BYVAL x2, BYVAL y2, BYVAL w2, BYVAL h2)

The function performed depends on the value of `flag`

- |   |            |   |
|---|------------|---|
| 0 | FMD_START  | does not actually do anything on current versions of the operating system, but should be called before making a series of <code>form_dial</code> calls. |
| 1 | FMD_GROW   | draws a box expanding from the rectangle given by (x1,y1,w1,h1) to that given by (x2,y2,w2,h2). This is similar to <code>graf_growbox</code> .          |
| 2 | FMD_SHRINK | draws a box shrinking from (x2,y2,w2,h2) to (x1,y1,w1,h1) This is similar to <code>graf_shrinkbox</code> .  |
| 3 | FMD_FINISH | sends messages to re-draw the screen for any windows inside the rectangle given by (x2,y2,w2,h2).   |

### FUNCTION form\_alert(BYVAL default, BYVAL alert\$)

Displays an alert on the screen and lets the user interact with it. The default button is given by the `default` parameter and is 1 for the first button, 2 for the second etc and 0 if there is no default button. The value returned is the number of the button selected. The screen is restored by the AES; there is no need to re-draw the screen. `alert$` has the form:

[icon][message][button1|button2.....]

`icon` is the number of the icon to display:

- |   |            |
|---|------------|
| 0 | no icon    |
| 1 | ! icon     |
| 2 | ? icon     |
| 3 | STOP icon. |

`message` is the text to display in the alert box; it should not exceed 200 characters and should contain | (vertical bar) characters to delimit the lines. `button1` and `button2` are the text for the buttons. There may be up to three buttons; the text for each can not exceed 20 characters.

If the `text$` parameter does not conform to the above rules the machine may crash.

### FUNCTION form\_error(BYVAL errorcode)

This displays the error message box corresponding to the GEMDOS `errorcode` parameter unless this is greater than 63 when the call is ignored. The value returned is the button that was clicked on. This is always 1 on current versions of the operating system.

### SUB form\_center(BYVAL tree&, x, y, w, h)

This routine returns the co-ordinates (x,y) and width `w` and height `h` of a dialog box object tree `tree&` that are needed to centre the object on the screen ready for use in `form_dial` etc. The tree itself is also modified.

## GEMAES: Graphics

### SUB graf\_rubberbox(BYVAL x, BYVAL y, BYVAL minw, BYVAL minh, outw, outh)

Lets the user drag a rubber box starting at (x,y) with a minimum width and height of `minw` and `minh`. The current width and height when the user lets go of the mouse button are returned in `outw` and `outh`.

### SUB graf\_dragbox(BYVAL w, BYVAL h, BYVAL startx, BYVAL starty, BYVAL boxx, BYVAL boxy, BYVAL boxw, BYVAL boxh, outx, outy)

lets the user drag a box of fixed size (w,h) starting at `startx,starty` returning the final position in `outx` and `outy`. (The box will always be contained within the rectangle specified by `boxx,boxy,boxw` and `boxh`).

### SUB graf\_movebox(BYVAL w, BYVAL h, BYVAL x1, BYVAL y1, BYVAL x2, BYVAL y2)

draws a box of width `w` and height `h` moving from (x1,y1) to (x2,y2).

```
SUB graf_growbox(BYVAL x1, BYVAL y1, BYVAL w1,
  BYVAL h1, BYVAL x2, BYVAL y2, BYVAL w2,
  BYVAL h2)
```

draws a box outline expanding from the rectangle given by x1, y1, w1, h1 to that given by x2, y2, w2, h2.

```
SUB graf_shrinkbox(BYVAL x1, BYVAL y1, BYVAL
  w1, BYVAL h1, BYVAL x2, BYVAL y2, BYVAL
  w2, BYVAL h2)
```

draws a box outline shrinking from the rectangle given by x2, y2, w2, h2 to that given by x1, y1, w1, h1. Note that in both graf\_growbox and graf\_shrinkbox the larger rectangle is second.

```
FUNCTION graf_watchbox(BYVAL tree&, BYVAL
  object, BYVAL instate, BYVAL outstate)
```

changes a box while the user moves it in and out of the box specified by object in the tree tree&. The mouse button *must* be down when this routine is called and it returns 0 if the mouse is outside the box when the user releases the button and 1 if it is inside. instate and outstate give the box's ob\_state field when the pointer is inside and outside of the box respectively.

```
FUNCTION graf_slidebox(BYVAL tree&, BYVAL
  parent, BYVAL object, BYVAL vertical)
```

Lets the user move the box given by the object from the tree tree& within its parent box given by parent. If vertical=1 then the movement is vertical; if vertical=0 then it is horizontal. The value returned gives the position of the inner box when the user releases the mouse button and ranges from 0 to 1000 with 0 being top or left depending on the value of vertical. The mouse button *must* be down when this routine is called.

```
FUNCTION graf_handle(wcell, hcell, wbox, hbox)
```

returns the current GEM VDI handle that the AES is using and the width and height of the character cells of the system font in pixels in wcell and hcell; together with the size of a box large enough to enclose a character in this font in wbox and hbox.

```
SUB graf_mouse(BYVAL flag, BYVAL user_form&)
```

This call changes the mouse form that is displayed according to the value in flag as described in the following table:

|     |  |
|-----|--|
| 0   | arrow  |
| 1   | text cursor (vertical bar)   |
| 2   | busy bee   |
| 3   | pointing finger  |
| 4   | extended fingers   |
| 5   | thin cross hair  |
| 6   | thick cross hair   |
| 7   | outline cross hair   |
| 255 | user defined mouse form given by the buffer pointed to by user_form& |
| 256 | hide mouse   |
| 257 | show mouse   |

The hide and show the mouse calls above have the disadvantage that, if the wind\_update calls are not used correctly, then the machine may crash or loose the ability to use menus. The v\_hide\_c and v\_show\_c GEM VDI calls do not suffer from this problem; however we have experienced the AES missing mouse clicks after certain combinations of these calls.

```
SUB graf_mkstate(x, y, state, kstate)
```

returns the current mouse position in (x,y), the current mouse button state and the state of the Shift, Alt and Ctrl keys. In state bit 0 is the left button and bit 1 is the right button. A value of 0 means the button is up and 1 means the button is down. kstate bits are as follows:

| Mask | Key         |
|------|-------------|
| 1    | right Shift |
| 2    | left Shift  |
| 4    | Ctrl        |
| 8    | Alt         |

As usual a bit is 0 if the key is up and 1 if the key is down.

## GEMAES: Scrap Directory

```
FUNCTION scrp_read(directory_name$)
```

Reads the current scrap directory into the string variable parameter directory\_name\$ and returns 0 if an error occurred. Returns non-zero if no error occurred.

## FUNCTION scrp\_write(BYVAL directory\_name\$)

Sets the current scrap directory to the string parameter `directory_name$` and returns 0 if an error occurred. Returns non-zero if no error occurred.

## GEMAES: File Selector

### SUB fsel\_input(path\$, name\$, ok)

Displays and lets the user use the standard GEM AES file selector. The initial and updated values of the drive and path are passed in `path$` and the initial and final file name are passed in `name$`. `name$` should be a maximum of 12 characters long. `ok` is returned as 0 if the user clicked on **Cancel** and 1 otherwise. If an error occurs `ok` is returned as 0 and the strings are not updated.

### SUB fsel\_exinput(path\$, name\$, ok, BYVAL prompt\$)

This displays and lets the user interact with the extended GEM file selector, whilst displaying a message to indicate the action about to be taken (e.g. **Save File**).

The parameters of this call are the same as for `fsel_input` except for the extra `prompt$` parameter. This string (which may be up to 30 characters long) is displayed instead of the **Item Selector** message.

This operating system call was added in Rainbow TOS (AES version 1.30).

## GEMAES: Windows

### FUNCTION wind\_create(BYVAL kind, BYVAL x, BYVAL y, BYVAL w, BYVAL h)

Creates a window with a maximum size given by the rectangle of width `w` and height `h` with top left (`x,y`) and returns the window's handle as the result of the function or a negative value if no windows are available.

`kind` specifies the components present in the window with the following bit masks for the individual components:

|      |         |                              |
|------|---------|------------------------------|
| 1    | NAME    | title bar with name          |
| 2    | CLOSE   | close box                    |
| 4    | FULL    | full box                     |
| 8    | MOVE    | can be moved                 |
| 16   | INFO    | information line below title |
| 32   | SIZE    | size box                     |
| 64   | UPARROW | upwards arrow                |
| 128  | DNARROW | down arrow                   |
| 256  | VSLIDE  | vertical slider              |
| 512  | LFARROW | left arrow                   |
| 1024 | RTARROW | right arrow                  |
| 2048 | HSLIDE  | horizontal slider            |

This call does *not* display the window; use `wind_open` (see below) to do this. It is *imperative* that the window is deleted before your program exits using `wind_delete` otherwise the window will not be deleted until you return to the Desktop; if you do this too often you will find that the system runs out of windows. We strongly recommend that you use the **WINDOW** commands built into HiSoft BASIC as these will always be closed and deleted as long as the program does not finish with a 68000 exception (bombs).

### FUNCTION wind\_open(BYVAL handle, BYVAL x, BYVAL y, BYVAL w, BYVAL h)

Displays the window with the given `handle` with an initial position of (`x,y`) and width `w` and height `h`. The co-ordinates include the window's border. The window handle is obtained from the `wind_create` call (see above). This routine returns 0 if there was an error ; 1 otherwise.

### FUNCTION wind\_close(BYVAL handle)

Closes the window with the given `handle`. This means that the window is not displayed but may be displayed again using `wind_open`. To delete a window you must close it and then call `wind_delete` (see below). This routine returns 0 if there was an error ; 1 otherwise.

### FUNCTION wind\_delete(BYVAL handle)

Deletes the window with the given `handle`. This means that the memory allocated to the window and the window handle are freed. If the window is displayed you *must* close it using `wind_close` (see above) first. This routine returns 0 if there was an error ; 1 otherwise.



FUNCTION wind\_get(BYVAL handle, BYVAL sort, x, y, w, h)

This call returns information about the window handle depending on the value of sort. This routine returns 0 if there was an error; 1 otherwise. For most values of sort x, y, w and h specify a rectangle as usual. In the table below the exceptions to this are noted. The names in this table are included in the GEMAES.BH file prefaced with WF\_.

| Sort | Name      | Action   |
|------|-----------|--|
| 4    | WORKXYWH  | the current work area of the window is returned.   |
| 5    | CURRXYWH  | the current size of the window including borders.  |
| 6    | PREVXYWH  | the co-ordinates of the previous window including borders.   |
| 7    | FULLXYWH  | the maximum size of the current window including borders.  |
| 8    | HSLIDE    | x contains the current position of the horizontal slider between 1 and 1000. 1 is the left most position.  |
| 9    | VSLIDE    | x contains the current position of the vertical slider between 1 and 1000. 1 is the top most position.   |
| 10   | TOP       | x contains the handle of the top (active) window.  |
| 11   | FIRSTXYWH | the co-ordinates of the first rectangle in the window's rectangle list.  |
| 12   | NEXTXYWH  | the co-ordinates of the next rectangle in the window's rectangle list.   |
| 15   | HSLSIZE   | x contains the size of the horizontal slider relative to the horizontal scroll bar (1 to 1000) or -1 if the default square box.                      |
| 16   | VLSIZE    | x contains the size of the vertical slider relative to the vertical scroll bar (1 to 1000) or -1 if the default square box.                          |
| 17   | SCREEN    | x and y give the address of the internal to the AES alert buffer and w and h give the length of this buffer. x and w are the most significant words. |

FUNCTION wind\_set(BYVAL handle, BYVAL sort, BYVAL x, BYVAL y, BYVAL w, BYVAL h)

This call sets the various attributes of the window according to the value of sort. This routine returns 0 if there was an error; 1 otherwise. For most values of sort x, y, w and h specify a rectangle as usual. Exceptions are noted under the particular sort. The following values of sort are the same as for wind\_get: CURRXYWH, HSLIDE, VSLIDE, TOP, HSLSIZE, VLSIZE. In addition the following are available:

| Sort | Name    | Action  |
|------|---------|---|
| 2    | NAME    | x, y are the address of the name of the window. x is the most significant word of the address.  |
| 3    | INFO    | x, y are the address of the text in the information line of the window. x is the most significant word of the address.  |
| 14   | NEWDESK | x, y are the address of an object tree containing the new GEM desktop for the AES to draw. w is the first object to draw in the tree. If x and y are both 0 then the desktop is reset to the default one. |

*Warning:* Be careful when passing strings to the AES using NAME and INFO as above, because these are not copied by the AES and thus if HiSoft BASIC performs a garbage collect the string may move. The solution is to store the string in a numeric array that is dimensioned before any arrays that are REDIMmed or ERASEd.

The following will pass the address stored in the variable x& as two integers:

```
PEEKW(VARPTR(x&)), PEEKW(VARPTR(x&)+2)
```

FUNCTION wind\_find(BYVAL x, BYVAL y)

returns the handle of window that is under the co-ordinate (x,y). 0 is returned if there is no window under (x,y).

FUNCTION wind\_update(BYVAL sort)

This routine can be used to stop the user using menus, moving windows etc. whilst the application is outputting to the screen or when the application wants to do its own tracking of the mouse. These routines should be called strictly in pairs. If you call END\_MCTRL more times than BEG\_MCTRL the machine may hang.

The different sorts are:

- 1 **BEG\_UPDATE** tells the operating system that the application is about to update the window and will wait until menus are not down before doing this. You should call this routine before writing to a window with the VDI.
- 0 **END\_UPDATE** tells the operating system that the application has finished updating the window and that the user may pull down menus once more. Should be called after you have called the VDI if you called this routine with **BEG\_UPDATE**.
- 3 **BEG\_MCTRL** tells the operating system that the application is performing all mouse control itself and the AES will not let the user pull-down menus or click on windows.
- 2 **END\_MCTRL** tells the operating system that the application has finished doing its own mouse control and so the AES will let the user, once more, pull down menus and click on close boxes etc. Must always be called if you have called this routine with **BEG\_MCTRL** beforehand.

This function returns 0 if there was an error ; 1 otherwise.

**FUNCTION** **wind\_calc**(BYVAL type, BYVAL kind,  
BYVAL x, BYVAL y, BYVAL w, BYVAL h,  
xout, yout, wout, hout)

Calculates the work area of a window in xout, yout, wout, hout given a border of a window in x, y, w, h and the sort of window in kind if type=1. If type=0 then the border area is calculated given the work area. kind is defined under **wind\_create**. This routine returns 0 if there was an error ; 1 otherwise.

#### **SUB** **wind\_new**

closes and deletes all windows, flushes all window buffers and returns to standard mouse usage including the **wind\_update** count.

This is the function that is used by the Desktop to tidy up after an application quits and so should be used if your application needs to run a possibly badly behaved program. Unfortunately this call is only available on AES version 1.30 (Rainbow TOS) and above, so that it cannot be used by lazy programmers to return to a fixed state!

## **GEMAES: Resource Files**

### **FUNCTION** **rsrc\_load**(BYVAL file\_name\$)

Loads the resource file called file\_name\$ into memory. The memory is allocated from GEMDOS. Normally only a limited amount of memory is returned to GEMDOS by the BASIC program but you can change this by using the L (leave) and K (keep) compiler options. See *Appendix B* and *Chapter 3* of the *User Manual*. Resource files are created by resource construction set programs such as HiSoft WERCS. This routine returns 0 if there was an error ; non-zero otherwise.

### **FUNCTION** **rsrc\_free**

Frees the memory used by **rsrc\_load**. This routine returns 0 if there was an error; 1 otherwise.

### **FUNCTION** **rsrc\_gaddr**(BYVAL type, BYVAL index, addr&)

Returns the address of a data structure from the resource file loaded using **rsrc\_load** in addr&. index is the index of the data structure in the object tree (returned by the resource construction set) and type gives the type of the object:

|    |            |  |
|----|------------|--|
| 0  | tree       |  |
| 1  | OBJECT     |  |
| 2  | TEDINFO    |  |
| 3  | ICONBLK    |  |
| 4  | BITBLK     |  |
| 5  | string     |  |
| 6  | image data |  |
| 7  | ob_spec    |  |
| 8  | te_ptext   |  |
| 9  | te_ptmplt  |  |
| 10 | te_pvalid  |  |
| 11 | ib_pmask   |  |
| 12 | ib_pdata   |  |
| 13 | ib_ptext   |  |
| 14 | bi_pdata   |  |
| 15 | ad_frstr   | the address of a pointer to a free string (i.e with INDIRECT set in its ob_flags field). |
| 16 | ad_frmng   | the address if a pointer to a free image. (i.e with INDIRECT set in its ob_flags field). |

**FUNCTION rsrc\_saddr(BYVAL type, BYVAL index, BYVAL addr&)**

Sets up the address of a free string or image given **type** and **index** as for **rsrc\_gaddr** to be **addr&**. **type** must be 15 or 16. This routine returns 0 if there was an error ; non-zero otherwise.

**SUB rsrc\_obfix(BYVAL tree&, BYVAL object)**

converts the object in **tree&** location from character co-ordinates to pixel co-ordinates. Character co-ordinates are used in resource files to store the co-ordinates of object so that they do not depend on the physical characteristics of the device, in particular on the ST's screen mode. Normally this conversion is performed automatically by the **rsrc\_load** call. Character co-ordinates have the character position in the least significant byte of the word and the positive or negative pixel offset in the most significant byte.

## **GEMAES: Shell Routines**

**FUNCTION shel\_read(cmd\$, tail\$)**

returns the file that the application was loaded from in **cmd\$** and the command line tail that was used in **tail\$**. The command tail is also available from the **COMMAND\$** function. This should be called before performing a **rsrc\_load** call, because otherwise the name of the resource file will be returned as **cmd\$**. This routine returns 0 if there was an error ; non-zero otherwise. If this is used from a program that is being developed interactively **HBASIC.PRG** is the current file.

**FUNCTION shel\_find(file\$)**

searches for the file named **file\$** in the current directory and the search path and returns this directory in **file\$.file\$** is therefore modified by the call. Note that if the file is found in the current directory this does not include the drive specifier. This routine returns 0 if there was an error; non-zero otherwise.

**SUB shel\_get(BYVAL buff&, BYVAL length)**

reads the AES's internal shell buffer (the RAM version of the **DESKTOP.INF** file) into the buffer at **buff&**; **length** bytes will be read. The buffer should be at least 4192 bytes long to accommodate for TOS versions later than AES version 1.40 (Rainbow TOS).

**SUB shel\_put(BYVAL buff&, BYVAL length)**

writes into the AES's internal shell buffer (the RAM version of the **DESKTOP.INF** file) from the buffer at the given address. **length** bytes will be written. The length must not be greater than 1024 bytes for AES versions prior to 1.40 (Rainbow TOS) or 4192 bytes for later TOS's. If you write a new buffer to the AES, you must place a single Ctrl-Z (26 decimal) to indicate the end of the buffer.

**FUNCTION shel\_write(BYVAL ex, BYVAL gr, BYVAL over, BYVAL commandname\$, BYVAL commandline\$)**

run another program when this application has finished. The **ex** and **over** parameters must be 1 in all current versions of the operating system. The **gr** parameter specifies whether the program to be run is a .TOS (or .TTP) program (use 0 for this parameter) or a GEM (i.e. .PRG or .APP) program.

The **commandname\$** parameter specifies the complete filename (including extension) of the program to be run. The **commandline\$** parameter specifies the command tail

**SUB shel\_envrn(env\$, BYVAL name\$)**

Searches the GEMDOS environment string for a particular variable and returns the result in **env\$.name\$** should include the = sign. Unfortunately with current versions of the operating system there is no easy way to set the variables. However if you use "PATH=" this will return a pointer to the current path. This can be modified by the Saved! by HiSoft desk accessory.

## **Menu Library**

**FUNCTION Menu&(BYVAL menu\_string\$)**

This routine creates an object tree containing a menu and displays it. The value returned is the address of the object tree for use in subsequent GEM AES calls.

An example of the format of the string is as follows:

```
menu$= "[ Desk | About HiSoft BASIC ]"
menu$=menu$+ "[ File | New \ Load \ (-----\ Save"
]" menu$=menu$+ "[ Search | Find ]"
```

Using this string will display a menu that is similar to part of the HiSoft BASIC menu bar. Each menu title and its items are enclosed in square brackets [ and ]. There is a vertical bar (|) after each menu title and the individual items are separated by back-slashes (\). For grey items precede the text with a single open parenthesis (. The first menu is always the desk title (normally Desk); the currently loaded desk accessories will automatically be added by the AES.

We recommend that you precede each menu item with two spaces and have a least one space after the item. Menu titles should have one space before and after them.

The number of each item is calculated as follows. Assuming we have n menu titles including Desk then the number of the desk menu title is 3, the first user title 4 ... the last user n+2. The About.. item is n+5, the first desk accessory (if present) is n+7, the sixth is n+12, and the first item in the first user menu is n+14. The second item is n+15 and so on. Grey items are included in this counting as are the menu titles. So if the last item on one menu is 21 the first on the next menu is 23.

Thus in the example above n=3.

|        |   |
|--------|---|
| Desk   | 3 |
| File   | 4 |
| Search | 5 |

Items:

|       |    |                             |
|-------|----|-----------------------------|
| About | 8  | (3+5)                       |
| New   | 17 | (3+14)                      |
| Load  | 18 |                             |
| Save  | 20 | skip one for the grey ----- |
| Find  | 22 | skip one for Search title   |

Obviously when you add or remove an item in a menu these numbers will change. It is therefore best to use the CONST construct to give the items readable names. This has the advantage of making your program more readable and it is much easier to change one constant rather than several.

The About.. message may not be more than 20 characters long (otherwise GEM will print them outside the box of the menu).

The memory for the menu tree is allocated using GEMDOS; subsequent MENU calls will free the space currently used. It is possible for the operating system memory to run out; see the options section for how to modify the amount a RAM allocated to this. This function also makes a copy of string on the stack whilst executing.

Menus created with this function will be "removed" when the program terminates.

## Chapter 3

# HiSoft GEM Toolbox

## Reference

The HiSoft GEM Toolbox (HGT) is a collection of BASIC source code modules that, together, allow you to access the Graphics Environment Manager (GEM) within your programs, quickly and easily.

We provide you with the source code to these routines for two reasons; firstly so that you can use just the parts that you need for your particular application and secondly so that you can appreciate how GEM works.

Normally you will use HGT by REM \$include the code that you need.

For any particular task you will not use all of the HGT modules although it is certain that you will need TOOLBOX.BAS since this contains all the initialisation code for the toolbox together with many of the global variables used. It is therefore a good idea to use a pre-tokenised version of TOOLBOX.BAS to cut down the compilation time; we have provided such a file for you (TOOLBOX.T in the HGT folder).

You can, of course, pre-tokenise any combination of the toolbox routines to speed up your development time. You do this by writing a small BASIC program that REM \$includes all the toolbox modules that you want to tokenise and then choosing Dump tokens from the Program menu.

We will now discuss each HGT routine in more detail, grouping them by their particular purpose.

Some routines must be supplied by you and these are marked with a sidebar, like the one alongside this paragraph. You will be told what this routine should do and you can find examples of different user routines in the HGT example programs.

## General control routines

The following modules are essential to most HGT programs and are included in the TOOLBOX.BAS file.



**SUB StartProgram(resourcefile\$, BYVAL  
treenumber, BYVAL exititem)**

This sub-program should be invoked at the beginning of programs that are to use the toolbox menu, resource file and window facilities. This loads the resource file and uses the menu tree given by `treenumber` as the menu, with `exititem` specifying the Quit item on that menu. This routine also initialises the toolbox's file selector.

If you have a program that uses the window and file selector facilities without a menu, use "" for the `resourcefile$` string.

If you want to do your own handling of the Quit item, perhaps to request confirmation from the user, you should pass -1 as the exit item. To load a resource file without displaying a menu pass 0 as the `exititem` parameter.

Note that this routine will call `InitResourceFile` and `InitMenuSystem` for you, automatically, if required.

### **SUB HGTLLoop**

This procedure is responsible for the main work of the toolkit. It handles and responds to the keyboard, mouse and menus. You should call it after calling `StartProgram` and after your own initialisation code.

In turn, `HGTLLoop` will call your own routines `ProcessClicks`, `Processkeys` and `ProcessUserMenus` for you as appropriate. These sub-programs must be written by you as described below if your program allows mouse, keyboard or menu input respectively. Setting `finished_flag` to -1 within one of these routines will cause `HGTLLoop` to be exited.

### **SUB StopProgram**

This should be called when your program terminates; it frees the system resources that you have used within the toolbox.

### **SUB InitResourceFile(name\$)**

Loads the resource file with the given name. If the resource file can't be loaded (or there is too little free GEMDOS RAM to load it) an alert is displayed and the program quits. This is normally called for you by `StartProgram`.

### **SUB BusyBee(BYVAL Busy)**

If `Busy` is true (non-zero) then the mouse is displayed as a 'busy bee'; if it is 0 then it is displayed as an arrow.

**DECLARE SUB ProcessClicks(BYVAL clicks,  
BYVAL kstate, BYVAL x, BYVAL y)**

This sub-program should be supplied by you if you want to process mouse clicks yourself. If you set the `clicks_enabled` global variable to a non-zero value before invoking `HGTLLoop`, `ProcessClicks` will be called when the user single or double clicks with the mouse. The `clicks` parameter gives the number of clicks (1 or 2), the `kstate` parameter gives the state of the keyboard Shift keys as per `evnt_multi` and the `x` and `y` parameters passes the mouse position when the click was detected.

### **SUB Debug(BYVAL a\$)**

`Debug` can be used to display debugging messages on the screen in cases where using `PRINT` either isn't possible because you don't have a window open or will corrupt the standard output from your program. After displaying your message `Debug` waits for you to press a key before continuing in case the message is obscured. This routine uses the BIOS for output so it doesn't look very pretty.

The easiest way to display numeric data with `Debug` is to use one of the `FORMATx$` functions.

### **SUB AbortProgram(a\$)**

This procedure displays its argument message using `Debug` and then stops the program using `StopProgram`.

**SUB ProcessKeys(BYVAL ascii\$, BYVAL  
scan\_code, BYVAL kstate)**

This routine should be supplied by you if you wish to process key presses yourself (i.e. not merely menu keyboard shortcuts). Setting the global variable `keys_enabled` to a non-zero value will cause `ProcessKeys` to be called. This routine is *not* called if a menu keyboard shortcut is used.

The `ascii$` parameter gives the ASCII value of the character that has been selected. While this is suitable for ordinary characters and control sequences, in order to detect Alt combinations, function keys and special keys you will need to use the `scan_code` parameter. If you want to detect the state of the Shift keys directly, use the `kstate` parameter.

## **Object Handling routines**

The following object handling routines are part of the main toolbox, not least because they are used by the other parts of the toolbox. They are therefore in `TOOLBOX.BAS`.

You need to understand about the tree structure of a resource file before these routines are of much use to you. Read the *WERCS* chapters and, particular, the *Tree Structure* sections under *Programming with Resources* in *Chapter 4* where the concepts used below are explained.

#### SUB SelectTreeAddr(BYVAL t&)

Selects a new tree given the address of the root.

#### SUB SelectTree(BYVAL treeno)

Selects a new tree given the tree number from the resource file. You must have loaded a resource file using *StartProgram* or *InitResourceFile* to call this routine and you should use the constants in the .BH file that is created by *WERCS*.

#### FUNCTION ObjectAddr&(BYVAL object)

Returns the address of the *object* in the current tree. *object* is the index as given by *WERCS* (0, 1, ...). This function is published for completeness - you should not need to use it since all the subsequent functions need the *object number* rather than its address within the tree.

#### FUNCTION Getob\_head(BYVAL object)

Returns the value of the *ob\_head* field in the given *object* of the currently selected tree.

#### FUNCTION Getob\_tail(BYVAL object)

Returns the value of the *ob\_tail* field in the given *object* of the current tree.

#### FUNCTION Getob\_spec&(BYVAL object)

Returns the value of the *ob\_spec* field in the given *object* of the current tree.

#### FUNCTION Getob\_state(BYVAL object)

Returns the value of the *ob\_state* field in the given *object* of the current tree.

#### FUNCTION Getob\_flags(BYVAL object)

Returns the value of the *ob\_flags* field in the given *object* of the current tree.

#### FUNCTION Getob\_type(BYVAL object)

Returns the value of the *ob\_type* field in the given *object* of the current tree.

#### FUNCTION Getob\_next(BYVAL object)

Returns the value of the *ob\_next* field in the given *object* of the current tree.

#### FUNCTION Getob\_x(BYVAL object)

Returns the value of the *ob\_x* field in the given *object* of the current tree.

#### FUNCTION Getob\_y(BYVAL object)

Returns the value of the *ob\_y* field in the given *object* of the current tree.

#### FUNCTION Getob\_width(BYVAL object)

Returns the value of the *ob\_width* field in the given *object* of the current tree.

#### FUNCTION Getob\_height(BYVAL object)

Returns the value of the *ob\_height* field in the given *object* of the current tree.

#### SUB Setob\_state(BYVAL object, BYVAL value)

Sets the value of the *ob\_state* field in the given *object* of the current tree.

#### SUB Setob\_flags(BYVAL object, BYVAL value)

Sets the value of the *ob\_flags* field in the given *object* of the current tree.

#### SUB Setob\_x(BYVAL object, BYVAL value)

Sets the value of the *ob\_x* field in the given *object* of the current tree.

#### SUB Setob\_y(BYVAL object, BYVAL value)

Sets the value of the *ob\_y* field in the given *object* of the current tree.

#### SUB Setob\_width(BYVAL object, BYVAL value)

Sets the value of the *ob\_width* field in the given *object* of the current tree.

#### SUB Setob\_height(BYVAL object, BYVAL value)

Sets the value of the *ob\_height* field in the given *object* of the current tree.

**SUB Exclob\_state(BYVAL object, BYVAL flag\_mask)**

Removes a mask (e.g. mask\_selected, mask\_disable) from the ob\_state field in the given object of the current tree.

**SUB Inclob\_state(BYVAL object, BYVAL flag\_mask)**

Includes a mask (e.g. mask\_selected, mask\_disable) in the ob\_state field in the given object of the current tree.

**FUNCTION Curob\_state(BYVAL object, BYVAL flag\_mask)**

Returns non-zero if the mask (e.g. mask\_selected, mask\_disable) from the ob\_state field is set.

**SUB Exclob\_flags(BYVAL object, BYVAL flag\_mask)**

Removes the mask (e.g. mask\_selectable, mask\_exit) from the ob\_flags field in the given object of the current tree.

**SUB Inclob\_flags(BYVAL object, BYVAL flag\_mask)**

Includes the mask (e.g. mask\_selectable, mask\_exit) in the ob\_flags field in the given object of the current tree.

**FUNCTION Curob\_flags(BYVAL object, BYVAL flag\_mask)**

Returns non-zero if a mask (e.g. mask\_selectable, mask\_exit) from the ob\_flags field is in the given object of the current tree.

**SUB Sette\_ptext(BYVAL object, BYVAL newted\$)**

Sets the string data in the tedinfo record of the given object to be newted\$. Useful for changing the values that will be displayed in FText and FBoxText objects prior to calling HandleDialog.

**FUNCTION Gette\_ptext\$(BYVAL object)**

Returns the current value of the text field of the Tedinfo record of the given object. Useful for reading the values entered in FText and FBoxText objects prior to calling HandleDialog.

**FUNCTION Getob\_spec\$(BYVAL object)**




Returns the string pointer to be the ob\_spec field of the object as a BASIC string. Useful for String, Text and BoxText objects.

## Menu routines

These routines are part of the MENU.BAS file.

**SUB InitMenuSystem(BYVAL treenumber, BYVAL exititem)**

InitMenuSystem displays the menu given by treenumber from the resource file and scans that menu for any keyboard short cuts. The parameter exititem should be the object number of the item to be used to terminate the program; pass -1 as the exititem if you wish to handle the exit item yourself. You shouldn't normally need to call this routine directly as it is called by StartProgram.

When designing your menu using WERCS, the  symbol (character 1) should be used to indicate Shift keyboard shortcuts, the  symbol for Ctrl shortcuts and  (character 7) for Alt shortcuts. You may only use one of these modifiers per key and this modifier can be applied to any ASCII characters on the keyboard.

Menu keyboard short cuts can also be F1, F2, ... F10 and their Shift variants or one of the following special keys:

|                    |      |                  |
|--------------------|------|------------------|
| BS (for Backspace) | Clr  | Esc              |
| Help               | Home | Ins (for Insert) |
| Tab                | Undo |                  |

**DECLARE SUB ProcessUserMenus(BYVAL wind, BYVAL item, BYVAL title)**

This routine must be supplied by you if you have a menu and then call the toolbox HGTloop sub-program. It is called when the user selects a menu item.

The wind parameter is the AES handle of the top window so you can tell which of your windows the user currently has selected. The item parameter gives the object number of the item selected (as given by WERCS) and the title gives the object number of the title of this menu.

This routine is also called if the user selects one of the shortcuts from the menu, just as if they had actually used the mouse.

Note that, when the exit item that was passed to `InitMenuSystem/StartProgram` is called, `HGTloop` will return without `ProcessUserMenus` being called.

## Dialogs and Alerts

The following routines are in the `DIALOG.BAS` file. With the exception of the alert routines, the tree must have been selected using `SelectTree` or `SelectTreeAddr` before you use these sub-programs.

**FUNCTION** `Newform_alert`(BYVAL num, BYVAL button)

**FUNCTION** `Newform_alertaddr`(BYVAL button, BYVAL addr&)

These are additional GEM calls that are required to use C-type strings (as loaded from resource files) for alerts. The `Newform_alert` routine takes the Alert's index from the resource file (as provided in the `.BH` file by `WERCS`) whereas the `Newform_alertaddr` routine takes the actual address of the string.

**FUNCTION** `HandleDialog`(BYVAL editnum)

A general routine to display a dialog box and handle interaction. The return result is the exit object number. The `editnum` parameter gives the text field where the cursor will be positioned initially; if you don't have any text fields then pass the value 0.

`HandleDialog` centres the form and draws appropriate growing and shrinking boxes for you. It also de-selects the form's exit button.

**SUB** `SetTedNumber`(BYVAL object, BYVAL value#)

Sets a `tedinfo` field to a numeric value. This is the equivalent to `Sette_ptext` for strings. The format used is the same as would be used if the value were printed as a double. For more control use the `FORMATx$` functions and `Sette_ptext`.

**FUNCTION** `GetTedNumber#`(BYVAL object)

Returns a numeric value from a `tedinfo` field. This is the equivalent to `Gette_ptext$` for strings.

**SUB** `SetButton`(BYVAL parent, BYVAL button)

Selects a particular radio button in a group, de-selecting all others in that group. The `parent` parameter gives the box containing the radio buttons while the `button` parameter gives the object to be selected; this is the object number within the tree.

**FUNCTION** `GetButton`(BYVAL parent)

Returns which button of a group of radio buttons is selected.

## Other object routines

This collection of routines is designed for people who are manipulating object trees directly in memory and the routines are contained in `OTHOBJEC.BAS`. You will need to be familiar with the structure of object trees before using them. As with the more basic routines, the tree to use must have been selected using `SelectTree` or `SelectTreeAddr`.

**FUNCTION** `GetParent`(BYVAL index)

Returns the object number of this object's parent or -1 if it is the root.

**SUB** `ForEachChild`(BYVAL parent, BYVAL procadr&)

`ForEachChild` is a powerful sub-program that lets you write other procedures to manipulate object trees yourself. `ForEachChild` handles the tree walking for you and you just need to write the sub-program that is executed for each child. Often this can be just a single line as with many of the commands that follow.

The procedure that is called should be parameterless and should be passed to `ForEachChild` using the `VARTPRS` function. The sub-program is called for each child of the object parent according to the tree structure.

The sub-program can read the following, globally-shared, variables:

|                       |   |
|-----------------------|---|
| <code>current</code>  | the object number in the tree of the current object.                            |
| <code>ithchild</code> | 0 for the first child, 1 for the second etc.                                    |
| <code>stop_now</code> | setting this to a non-zero value will stop any other children being considered. |

**FUNCTION** `HowManyChildren`(BYVAL item)

Returns the number of children that the given object has.

**FUNCTION** `WhichChild`(BYVAL item)

Returns which child of its parent tree `item` is, or -1 if `item` has no parent. 0 is returned for the first child, 1 for the second etc.

**FUNCTION** `IsChild`(BYVAL parent, BYVAL PossChild)

Returns -1 (true) if `PossChild` is a child of (or the same as) `Parent`.



## FUNCTION ObjectNumber(BYVAL par, i)

Returns the object number in the tree of the *i*th child of the object *par*. As usual 0 is used for the first child, 1 for the second etc.

# The Window system

This section describe the core of the toolbox's window system that is supplied in WINDOW.BAS. In general you probably won't need to call most of these directly as the higher level routines, described below under *Object*, *Image* and *Text* windows, will suffice.

In general when you use the toolbox you use AES handle numbers. However, internally, the toolbox has its own window numbers which are not the same as AES window handles or the same as HiSoft BASIC window handles. Most of your routines can just use the AES handle but we provide the ability to convert to/from the other types if you need to access the different window calls or you are writing your own window routines.

## FUNCTION WhichWindow(BYVAL AEShandle)

This function returns the Toolbox window handle corresponding to the AES window parameter that is supplied as a parameter.

## FUNCTION OpenAWindow(WindowName\$, BYVAL WindComp, BYVAL procaddr&, BYVAL slidermaxaddr&, BYVAL closeproc&)

Creates, opens a window and clears it to white with the given name and components and returns the corresponding toolbox window handle. If there was an error a HiSoft BASIC runtime error occurs.

*procaddr&* is a low-level re-draw procedure which is called when an area of the window needs to be re-drawn.

This should use the following global variables:

|              |  |
|--------------|--|
| commonx      | The rectangle to be re-drawn.                |
| commony      |  |
| commonw      |  |
| commonh      |  |
| commonhandle | The AES handle of the window to be re-drawn. |

Before the re-draw routine is called, the window will be cleared to white.

*slidermaxproc&* is the address of a sub-program to return the current maximum values for the window's sliders. The global variables that are set up before this routine is called are:

|              |  |
|--------------|--|
| commonhandle | The AES handle of the window to be re-drawn.                                       |
| commoni      | The index within the window system's internal tables (as returned by WhichWindow). |

The *slidermax* sub-program should return the maximum possible size of the scrollable area in pixels in the following variables:

|            |  |
|------------|--|
| commonmaxx | The maximum slider value in the x-direction. |
| commonmaxy | The maximum slider value in the y-direction. |

The *CloseProc&* parameter of *OpenAWindow* gives the address of the sub-program that is called just before the window is closed. This routine is passed the index of the current window in the window system's internal tables in *commoni*. This close routine is normally used to adjust any internal tables that are particular to a given window type.

The window components are as specified for the HiSoft BASIC WINDOW OPEN statement and for the AES *wind\_create* call

## SUB CloseAWindow(WindowHandle)

Close a window of the given AES handle. Note that this is passed a variable parameter which will be set to 0 on exit. This means that you can use the window's handle to detect whether that particular window is open or not.

## FUNCTION IsMyWindow(BYVAL WindowHandle)

Returns true (-1) if *WindowHandle* was opened using one of the Toolbox routines that open a window.

## SUB RenameWindow(BYVAL WindowHandle, WindowName\$)

This subprogram changes the title of the window whose AES handle is *WindowHandle* to be *WindowName\$*.

## SUB InitWindowSystem

This initialises the window system. It sets up the internal variables used by the window routines in the toolbox. Normally this will be called the first time that you invoke a routine that opens a window.

## Object Windows

These functions are in the OBJWIND.BAS file. This requires the WINDOW.BAS file to be included first.

```
FUNCTION OpenObjectWindowAddr(WindowName$,  
    BYVAL WindComp, BYVAL treeaddr&)
```

```
FUNCTION OpenObjectWindow(WindowName$, BYVAL  
    WindComp, BYVAL treeno)
```

These routines open a window and associate an object tree with that window that will be re-drawn automatically and can be re-sized and scrolled by the user. This gives an easy way to design control panel windows. To update the information to be displayed you can use the object family functions described above.

Both of these routines require the name and components of the window when creating a window. The OpenObjectWindow function is passed the tree number within the current resource file whereas the OpenObjectWindowAddr function is passed the address of the tree in memory.

## Image window routine

This routine is in the IMAGWIND.BAS file and requires both the WINDOW.BAS and IMAGE.BAS files to be included first.

```
FUNCTION OpenImageWindow(WindowName$,  
    WindComp)
```

Opens a scrolling window that displays an image - the image information is taken from the images() array. ImageWindows is the index within this array. The images() array and the actual image data will usually have been set up using

```
LoadPicture filename$,ImageWindows
```

See LoadPicture below for details of the types of images that may be loaded.

## Text windows

The following routines are in the TEXTWIND.BAS file which requires the WINDOW.BAS file.

```
SUB InitTextWindows (BYVAL MaxFiles, BYVAL  
    MaxLines)
```

This sets up the internal variables that are particular to text windows and dimensions the global arrays lines\$() and wlines().

The first of these arrays is where the actual text of windows is stored; lines\$(0,0) is the first line of the first text window, lines\$(0,1) is the second line of the first text window, lines\$(0,2) is the third line of the first text window and so on. lines\$(1,0) gives the first line of the second text window.

The maximum line number to display for the first text window is given by wlines(0), that for the second window in wlines(1) etc.

The MaxFiles parameter gives the maximum number of text windows that you may have open at once and MaxLines gives the maximum number of lines that may be displayed for each file.

The lines\$ and wlines() arrays require MaxFiles\*(MaxLines\*8+2) bytes of storage not including the actual text of the strings, so don't pass huge numbers to InitTextWindows.

InitTextWindows also sets the following variables, that can be modified by the user, to the values shown:

|              |     |
|--------------|-----|
| aligntext    | -1  |
| tabsize      | 8   |
| MaxTextWidth | 300 |

```
FUNCTION OpenTextWindow(WindowName$, BYVAL  
    WindComp)
```

Opens a scrolling window that displays text - the text is stored as lines in the lines\$(i,n) array where i+1 is the text file number. The number of lines of text to display is read from wlines(i). You must call InitTextWindows before using this routine.

The variable TextWindows gives the index in the lines\$() array. Thus you typically use TextWindows to get a window number, set up the corresponding entries in lines\$() and wlines() and call OpenTextWindow.

To change the text displayed you modify these arrays directly. Note that closing a text window will cause the index in the arrays for text windows opened after this one to decrease by one. The variables used are:

|  |  |
|--|--|
| <code>aligntext</code>                               | If this is non-zero then text windows will be set up so that the text is displayed on 8 pixel boundaries. This gives a much faster text display and is selected by default.  |
| <code>expandtabsflags</code><br><code>tabsize</code> | If <code>expandtabsflag</code> is non-zero then any Tab characters in the text are expanded on screen to the next <code>tabsize</code> boundary. Disabling this will give a slightly faster display. Expansion of tabs to 8 characters is selected by default. |
| <code>MaxTextWidth</code>                            | This is the value that the full width of the horizontal slider represents. It is 300 by default.   |

### SUB LoadTextFile(name\$, BYVAL index, lines)

This sub-program loads the file called `name$` into `lines$(index)`. The first line goes into `lines$(index,0)` the second into `lines$(index,1)` etc. The total number of lines read is then placed in `lines`. This is ideal for loading and displaying a text file in a window since all you need is:

```
LoadTextFile name$,textWindows,wolines(TextWindows)
texthandle=OpenTextWindow(name$,win_all-win_info)
```

### SUB ExpandTabs(sce\$,dest\$)

This sub-program expands any tabs in the string `sce$` and writes the expanded text to `dest$`.

The tab size is specified by the global variable `tabsize`.

## User Desktop routines

These routines are in the file `NEWDESK.BAS`.

### SUB SetDeskAddr(BYVAL newdesk&)

This sub-program sets the desktop pattern to be a particular object tree whose address is given by `newdesk&`. `SetDeskAddr` also ensures that the desktop is redrawn to show this change. To remove any custom desktop pass 0 as a parameter to this sub-program.

### SUB SetDesk(BYVAL treeno)

`SetDesk` lets you specify the desktop to be used by giving a tree number from your resource file. There's no need to set up the co-ordinates of the root object, `SetDesk` will do this for you automatically. To remove your custom desktop use a `treeno` value of -1.

## File Selector Routines

These routines are in the `FILESEL.BAS` file.

### FUNCTION FileSelect\$

This routine provides a 'quick-and-easy' interface to the GEM Item Selector - it simply returns the pathname that the user selected, so that you can use the file without any further processing. If the user selected Cancel a null string is returned.

The default path and file names are retained from call to call but if you wish to set these directly you can modify the global variables `fspath$` and `fname$`. You might wish to use a separate path when loading and saving the preferences file of your program or when requesting a particular type of file.

If the `fspath$` variable is null when `FileSelect` is called then `InitFileSelector` is called. Thus, in practice, the path used is the current directory when `FileSelect$` is first called and there's no need to call `InitFileSelector` directly.

### SUB InitFileSelector

This initialises the file selector path used by `FileSelect$` to be all files in the current directory and for there to be no default file.

## Running other programs

The following routines are in the `EXEC.BAS` include file.

### FUNCTION GetWholeEnvironment\$

This function returns the entire environment string of the current program so that it can be passed on to a child process, by the `Execute` sub-program for example.

### SUB Execute(nname\$,commlne\$)

Execute executes the command called nname\$ with a command line of commlne\$. It is 'polite' in that it removes any menu bar that the program has displayed and closes any windows that the program has created. It then re-displays the menu bar and windows when the child process terminates.

Note that this function does not delete any current open windows. Therefore, if you run a child process when you already have a number of windows open, the child won't have many windows available for its use. Also this function does not search the GEMDOS PATH environment variable or the AES path to find the child program.

## Image routines

These routines are in the IMAGE.BAS file.

### SUB LoadPicture(filename\$, BYVAL index)

This routine loads a GEM (.IMG), Degas (.PI1, .PI2, .PI3, .PC1, .PC2, .PC3) or Neochrome (.NEO) format picture file into RAM so that it can be subsequently displayed by the DisplayImage and OpenImageWindow routines.

The images(index...) will contain a GEMVDI MFDB structure that is used by the display routines, while the actual image data is loaded into memory that is allocated from GEMDOS memory.

You should ensure that the file that you are loading exists; if an image file is malformed or there is insufficient memory to load the image then the program will abort with a suitable message.

### SUB DisplayImage(BYVAL index, BYVAL x, BYVAL y)

This routine displays an image that has been loading using LoadPicture(filename\$,index) at screen co-ordinates (x,y). If you want to write your own custom image display routines, perhaps to display just a portion of an image, you can base them on this routine.

### SUB CloseImage(BYVAL index)

This frees the memory that was used by the image specified by index..

### SUB CloseAllImages

CloseAllImages frees the memory that was used by all the images that were loaded via LoadPicture.

## Desk Accessories

The file ACCTBOX.BAS is the equivalent of the main TOOLBOX.BAS but will let you write desk accessories.

Note that the caveats mentioned in the *Desk Accessory* appendix concerning memory management and screen resolution changes apply equally to the toolbox. ACCTBOX.BAS has exactly the same facilities as TOOLBOX.BAS except:

StartProgram, StopProgram, ProcessClicks, ProcessKeys, and all the menu facilities are not present but are replaced by:

### SUB StartAccessory(daname\$)

This is the equivalent to StartProgram and is passed the text to appear in the Desk menu for this desk accessory.

### SUB AccOpened

This is the routine that you must write which is called every time the desk accessory item for this DA is selected; the window system will already be initialised ready for use when this is called although you will need to call InitTextWindows if you are using TextWindows.

The variable AlreadyOpen is set to -1 if the desk accessory has a window open.



# Chapter 4

## WERCS Reference

---

This chapter contains detailed information on programming with resource files, compiling the WERCS example program and the file formats used by WERCS.

This section of the manual is designed to help you when programming with resource files that have been created by WERCS. The first section describes the objects and their attributes, whilst the second considers the programming aspects of these data structures. It concludes by discussing the WERCS' specific file formats.

### Objects

---

There are thirteen different types of objects as follows:

#### **Box**

A Box is a rectangle whose interior colour and fill pattern is controllable, as is its border thickness.

#### **BoxChar**

A BoxChar is a graphic box containing a single text character. It also has colour and border size attributes.

#### **BoxText**

Similar to Text objects (see later) but in addition surrounded by a border whose size and colour can be specified.

#### **Button**

A Button is displayed as a centred string of characters with a border. If the *default* flag is set, pressing the Return key in the standard form-handler is the same as clicking on the Button. A Default button is shown with a wider border. A new Button created with WERCS has its Selectable and Exit flags set.

## FBoxText

An FBoxText object is a formatted BoxText object; in addition to the normal Text attributes, it also has border attributes, a template and two extra strings for text entry. These extra strings are called the Template and Valid fields. The AES displays the Template string as if it were displaying any other type of text except that for every underline character it displays a character from the main text string. For example, a date field object might consist of a Text entry of:

011088

and a Template of

Date:\_\_\_/\_\_\_/\_\_\_

this would be displayed as

Date:01/10/88

if it were an FBoxText.

*Remember:* underline characters are entered as tildes (~).

The Valid string is used when the object is used as a Form using the GEM `form_do` command. The Valid string specifies which characters may be typed for each underline character in the Template string.

The different validation characters are:

|   |                                     |
|---|-------------------------------------|
| X | all characters allowed              |
| 9 | Only 0-9                            |
| A | A-Z and space                       |
| a | A-Z, a-z, 128-255 and space         |
| N | A-Z, 0-9 and space                  |
| n | A-Z, a-z, 0-9, 128-255 and space    |
| P | A-Z, a-z, 0-9, 128-255, \ : ? * . _ |
| p | A-Z, a-z, 0-9, 128-255, \ : _       |
| F | A-Z, a-z, 0-9, 128-255, : ? * _     |
| f | A-Z, a-z, 0-9, 128-255, _           |

In the above A-Z includes non-English capital letters. 128-255 means that all characters greater with value greater than 128 can be used, including lower case non-English letters and the £ sign.

You can use different validation characters in the same string if you wish. Thus for the date example above we would use the 9 character for all 6 character positions since the only characters allowed in dates are digits.

The most commonly used of these validation digits are probably X and 9. Note that if you wish to enter negative numbers you have to use X (otherwise the - sign would not be allowed).

Also the pathname options (P and p) are of limited value as a number of software producers sadly use illegal pathname characters such as - in their filenames. All but the X, F and f validation characters also have the undesirable feature with the first, 1.0, operating system ROMs of crashing the system when you press \_!

Note that whilst validation characters, P, p, F and f allow you to enter lower case letters these are echoed as the upper case equivalents.

If otherwise illegal characters are present in the Template string then the AES will skip past them if they are entered. With our date example typing / will skip to the next field even though / is not otherwise a valid character.

You should ensure that there are at least as many underlines in the Template string as there are characters in the main Text string; otherwise all of the latter will not be displayed. If you are intending to use this object as a Form *in situ* as normal, you should have the same number of characters in the Text string as there are underlines in the Template; if you don't observe this then if the user types a long string, the next string from your resource file will be corrupted. This restriction does not apply if you are intending to change the address of the Text field when the resource file is loaded.

So, in general, ensure that there are exactly the same number of characters in your Text string as there are underlines in your Template string.

Surprisingly, the Valid string does not have to contain a character for every underline in the Template string; if all the validation characters are the same then you can use just one. We have not seen this officially documented but it certainly works on all versions of the operating system we have used and can lead to considerably reduced resource file and memory usage if you have long strings.

If the first character in a text field is the at-sign (@), then `form_do` will display your string as underlines and place the cursor at the start of the string. Thus you can enter a blank string of n characters by typing, say ~, n-1 times, press cursor left until you are at the start of the string and then press @. The string will then disappear; but don't worry; it will be stored in your file ready for use.

## FText

Similar to a FBoxText (see above) but without border attributes.

## IBox

An IBox is a so-called *invisible box*, similar to a Box but hollow. It is only truly invisible if its border has a thickness of zero.

## Icon

This consists of two bit-map images, one for data and one for a mask. In addition a string of characters and a single character are also associated with it. Icons also have their own foreground and background colours.

## Image

An Image is a graphic bit-map with a foreground colour attribute. It differs considerably from an Icon; it has no mask (so cannot be distinguished on a patterned background), and no associated text or single character.

## ProgDef

This type of object is for advanced assembly language programmers only. It allows you to create your own types of object by supplying your own drawing routines. ProgDefs are displayed in WERCS as boxes with a diagonal line. ProgDefs are also known as *UserDefs*. We have seen the latter term used mainly in older documentation; a hangover perhaps from days when a Digital Research *user* was someone who wrote the assembly language to install CP/M on their computer.

## String

A String is a sequence of characters drawn in black and in the standard system font. If you require different sized or coloured text you should use one of the formatted text object types.

## Text

Actually graphic text; this is a sequence of characters that can be displayed in the system font or in a small font and can be left-, centre- or right-justified.

## Title

Objects of type Title are only used as Menu titles. Their use in other types of tree is not recommended; they have the same attributes as Strings.

## Flag Types

The different flag types for objects are as follows:

### Selectable

The Selectable flag is used in conjunction with the `form_do` AES routine. If this flag is set then if the user clicks on the object during a `form_do` call it will be highlighted and the Selected state bit will be set. If the Selected bit was already set, the object will be shown as normal and the Selected bit reset.

Thus setting this bit effectively turns any object into a Button without changing the appearance of the object. All Buttons that are to be used as such should have this bit set; this is the default when you create a new Button with WERCS.

### Default

The Default bit tells the AES that this is the default button of the form, i.e. the one which will be returned if the user types Return.

Normally this is used for Buttons but can also be used for other types. With Buttons the Default bit causes the object to be displayed with a wider border so that the user can see the default. For other objects there is no change in the screen display.

If the Default bit is set for an object you should normally also set the Selectable and Exit bits.

We do not recommend having more than one Default item in a form; the user, your program and the AES are likely to get confused.

### Exit

The Exit bit is used to indicate that clicking on this object will cause `form_do` to return to your program, with the index of this object as its result. If the Exit bit is not set the user can continue to edit the Form.

This bit can be used for any type of object, but only with Buttons is the size of the border increased to indicate to the user that this is an Exit Button. When you create a new Button using the Object menu this bit will be set.

The Selected bit should be set whenever the Exit bit is set.

## Editable

The Editable bit should only be set for the FText and FBoxText objects; this indicates that the user may edit the text in this field. If you set this bit for other types of objects the AES will mis-behave often causing the system to bomb. The fields in the TEDINFO structure used by the AES for FText and FBoxText objects must conform to strict rules as described under FBoxText.

There is no need to set other flags in conjunction with the Editable flag.

Do not use an Editable text field as the *last* object in a Form; all the current versions of the operating system will crash if you press cursor down when editing this field.

## Radio Button

The Radio Button bit is used to indicate that an object is one of a set of radio buttons. The objects need not be of type Button.

Every sibling of the object should have the Radio Button bit set; to ensure this you can use an IBox to surround just the objects that you wish to be Radio Buttons. Radio Button objects should have the Selectable flag set.

For an example of programming with Radio Buttons see the WTEST program.

## Touch Exit

The Touch Exit bit is used to tell the AES to exit `form_do` when the user moves the mouse pointer over an item and clicks on it. The exit occurs when the mouse button is pressed down (rather than released as in the case of the Exit flag). Touch Exit also differs from Exit in that the button need not be Selectable. When using `form_do`, if the user double-clicks on a Touch Exit object then the top bit of the return value will be set. Even if your program is not interested in double-clicks, you still need to mask off the top bit.

This flag may be used with any kind of object.

## Hide

The Hide bit is used to hide an object and all its children from the AES. This means that the object is not displayed by `objc_draw` and will not be found by `objc_find`. This is useful when you wish to remove part of a tree temporarily, without re-organising that tree. For example, WERCS itself uses this facility when drawing the Extras dialog box to ensure that only appropriate types of objects are shown.

If you have hidden an object using WERCS you cannot use the Hide command from the Flags menu to unhide it again because you cannot select it; instead select its parent and then use the UnHide Children command from the same menu instead.

## Flag States

The flag states for objects are as follows.

### Selected

If the Selected flag is set, it indicates that the object will be displayed highlighted. This bit is changed from 0 to 1 or from 1 to 0 if the object is Selectable when the user clicks on the appropriate object. Any type of object may have this bit set.

### Crossed

The Crossed bit causes the AES to draw a white diagonal cross through the object. If the object is Selected then the cross is displayed as black. This flag can be used on all objects except IBoxes.

### Checked

If the checked flag is set the AES will draw the object with a black tick mark, ✓, inside it with the tick in the top left corner. When the object is Selected the tick is shown in Black. The Checked flag may be used for any type of object including IBoxes.

### Disabled

If the Disabled flag is set for an object then it is shown greyed, that is, with less intense colour than normal. In addition, Disabled objects may not be Selected when using `form_do` or as part of a Menu even if they have the Selectable bit set. Note, though, that Disabled Editable fields may be edited!

### Outlined

If the Outlined bit is set then the object is drawn with a black box outside it. Note that this does not form part of the object as far as `objc_find`, for example, is concerned. This bit may be used with all types of objects.

### Shadowed

If the Shadowed bit is set for an object a shadow is drawn outside the object in the object's border colour; this includes Buttons. The Shadowed bit has no effect on objects without a border.

Selecting both Outlined and Shadowed attributes produces a messy display of the object and should be avoided.

## Object, Flags and States Summary

The following table shows which attributes change the appearance on screen for each type of object. Text Attr refers to the alignment and size of text:

|          | Fill<br>Pattern | Fill<br>Colour | Xparent/<br>Opaque | Border<br>Colour | Border<br>Size | Text<br>Colour | Text<br>Attr |
|----------|-----------------|----------------|--------------------|------------------|----------------|----------------|--------------|
| Box      | ✓               | ✓              |                    | ✓                | ✓              |                |              |
| BoxChar  | ✓               | ✓              |                    | ✓                | ✓              | ✓              |              |
| BoxText  | ✓               | ✓              | ✓                  | ✓                | ✓              | ✓              | ✓            |
| Button   |                 |                |                    |                  |                |                |              |
| FBoxText | ✓               | ✓              | ✓                  | ✓                | ✓              | ✓              | ✓            |
| FText    |                 |                | ✓                  |                  | ✓              | ✓              | ✓            |
| IBox     |                 |                |                    | ✓                | ✓              |                |              |
| Icon     |                 | ✓              |                    |                  |                | ✓              |              |
| Image    |                 |                |                    |                  |                | ✓              |              |
| ProgDef  |                 |                |                    |                  |                |                |              |
| String   |                 |                |                    |                  |                |                |              |
| Text     |                 |                | ✓                  |                  | ✓              | ✓              | ✓            |
| Title    |                 |                |                    |                  |                |                |              |

The following table shows the effect of particular flag/state sets for a number of the Flags. Remember that Selectable, Radio Button, TouchExit, Selected and Outlined may be used for all types of objects except Titles:

|          | Default | Exit | Editable | Crossed | Disabled | Shadowed |
|----------|---------|------|----------|---------|----------|----------|
| Box      |         |      | *        | ✓       |          | ✓        |
| BoxChar  |         |      | *        | ✓       | ✓        | ✓        |
| BoxText  |         |      | *        | ✓       | ✓        | ✓        |
| Button   | ✓       | ✓    | *        | ✓       | ✓        | ✓        |
| FBoxText |         |      |          | ✓       | ✓        | ✓        |
| FText    |         |      |          | ✓       | ✓        | ✗        |
| IBox     |         |      | *        | ✗       |          | ✓        |
| Icon     |         |      | *        | ✓       | ✓        | ✗        |
| Image    |         |      | *        | ✓       | ✓        | ✗        |
| ProgDef  |         |      | *        | ✓       | ✓        | ✗        |
| String   |         |      | *        | ✓       | ✓        | ✗        |
| Text     |         |      | *        | ✓       | ✓        | ✗        |
| Title    |         |      | *        | ✓       | ✓        | ✗        |

Key:

- ✓ Changes appearance of object and the behaviour of the AES.
  - Changes the behaviour of the AES but not the appearance.
  - ✗ Has no effect.
  - \*
- Causes the machine to crash with bombs.



# Programming with Resources

This section details the various data structures and object types, together with common AES programming algorithms. This section uses the standard names and offsets for the data structures and their components which are supplied in the header file `gemaes.bh`. It also describes the low-level functions used by the HiSoft GEM Toolbox for accessing these structures.

## Tree Structure

### OBJECT Structure

A tree is stored in memory as an array of objects. Each object has pointers to allow the AES to tree-walk as required. The structure is as follows:

|                        |                        |  |
|------------------------|------------------------|--|
| <code>ob_next</code>   | <code>word*</code>     | index of object's next sibling   |
| <code>ob_head</code>   | <code>word*</code>     | index of first child or -1 if none   |
| <code>ob_tail</code>   | <code>word*</code>     | index of last child or -1 if none  |
| <code>ob_type</code>   | <code>word*</code>     | object type (high byte is ignored by the AES and used for extended object numbers) |
| <code>ob_flags</code>  | <code>word†</code>     |  |
| <code>ob_state</code>  | <code>word†</code>     |  |
| <code>ob_spec</code>   | <code>longword*</code> | depends on object type   |
| <code>ob_x</code>      | <code>word†</code>     | X co-ordinate of object relative to parent (in pixels)                             |
| <code>ob_y</code>      | <code>word†</code>     | Y co-ordinate of object relative to parent (in pixels)                             |
| <code>ob_width</code>  | <code>word†</code>     | width of the object in pixels  |
| <code>ob_height</code> | <code>word†</code>     | height of the object in pixels   |

All the fields are present for all objects although the `ob_spec` field depends on the object type and is usually a pointer to another structure as described below.

Those items marked with † may be read and set using HGT routines of the form `Getob_xxx` and `Setob_xxx` and those with a \* may be read using `Getob_xxxx`. You must be sure to select the appropriate tree first via `SelectTree` or `SelectTreeAddr`. For example to read the type of the root object (object 0) of the current tree, you would use:

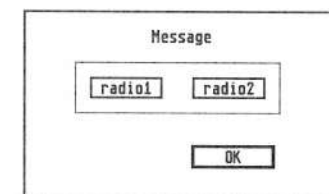
```
typ=Getob_type(0)
```

Similarly, to set the width of object 4 to be 32 pixels, you could use

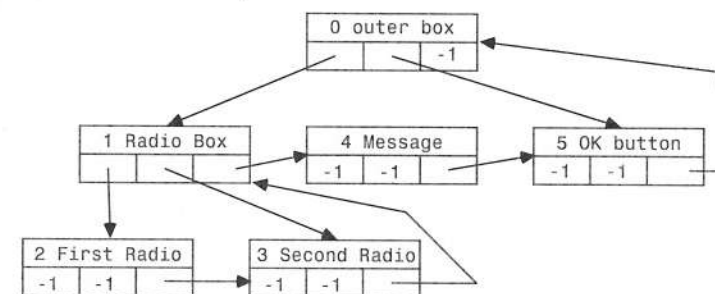
```
Setob_width 4,32
```

In a real program you would use the identifier from the `.BH` file created by WERCS rather than 4 itself.

When it is loaded into memory an object tree is like an array of records. The first object (with index 0) is called the *root object*. It is normally the outer Box of a dialog box. Each object in the tree has three fields called `ob_head`, `ob_tail` and `ob_next`. These hold integer values that dictate to the AES the structure of the tree. Fortunately you do not normally need to access these directly, WERCS does it for you. As an example, say we have a dialog box like this:



The tree structure this represents can be shown as:



where each box represents:

| obj index |      | name |  |  |
|-----------|------|------|--|--|
| head      | tail | next |  |  |

| <i>Index</i> | <i>ob_head</i> | <i>ob_tail</i> | <i>ob_next</i> | <i>name</i>  |
|--------------|----------------|----------------|----------------|--------------|
| 0            | 1              | 5              | -1             | outer box    |
| 1            | 2              | 3              | 4              | Radio Box    |
| 2            | -1             | -1             | 3              | First Radio  |
| 3            | -1             | -1             | 1              | Second Radio |
| 4            | -1             | -1             | 5              | Message      |
| 5            | -1             | -1             | 0              | OK button    |

Normally what is important with object trees is the tree structure not the order that the items are in memory. For example, don't assume that the first child immediately follows its parent.

|         |   |
|---------|---|
| ob_head | points to the first child (or is -1 if childless)                                     |
| ob_tail | points to the last child (or is -1 if childless)                                      |
| ob_next | points to the next sibling or, if there are no more siblings, to the object's parent. |

|   |        |    |
|---|--------|----|
| <b>Box</b>  | G_BOX; | 20 |
| The ob_spec field contains the colour word (low word) and border thickness (high word). |        |    |

**Text**                    G\_TEXT                    21

The ob\_spec field contains a pointer to a TEDINFO structure; the te\_pnext pointer within the structure points to the actual displayed text.

**BoxText**      G\_BOXTEXT      22

The `ob_spec` field contains a pointer to a `TEDINFO` structure; the `te_p_tst` pointer within the structure points to the actual displayed text.

The `ob_spec` field contains a pointer to the displayed string.



## Object Flags

The various flags in the `ob_flags` field have the following values as bits and as a hexadecimal mask:

| Name on Menu | Standard Name | Bit | Mask |
|--------------|---------------|-----|------|
| Selectable   | SELECTABLE    | 0   | 1    |
| Default      | DEFAULT       | 1   | 2    |
| Exit         | EXIT          | 2   | 4    |
| Editable     | EDITABLE      | 3   | 8    |
| Radio Button | RBUTTON       | 4   | 16   |
|              | LASTOB        | 5   | 32   |
| Touch Exit   | TOUCHEXIT     | 6   | 64   |
| Hide         | HIDETREE      | 7   | 128  |
|              | INDIRECT      | 8   | 256  |

The `LASTOB` bit is used by the AES to find the last object in an object tree; it is set for the last object and the last object alone. This bit is handled by WERCS for you but you may find it useful to access it if you write routines to manipulate trees in memory.

If the `INDIRECT` bit is set, the `ob_spec` field is treated as a *pointer* to the `ob_spec` field rather than the value itself. WERCS does not allow you to set this bit; if you need it then your program should set it and re-initialise the `ob_spec` field as required.

The GEMAES.BH defines these items as masks, pre-facing them with `mask_` and the HGT defines the following functions to manipulate these:

`Curob_flags` returns non zero if a given bit is set in an object and zero if it is not.

For example, `Curob_flags (i,mask_selectable)` returns true if the *i*th object in the current tree has the `SELECTABLE` bit set.

`Inclob_flags` set a given bit in an object. For example `Inclob_flags i,mask_default` sets the default bit for object *i*.

`Exclob_flags` resets a given in bit; thus `Exclob_flags i,mask_hidetree` will 'un-hide' object *i*.

## Object States

The following table gives the values as bits and masks of the `ob_state` field.

| Name on Menu | Standard Name | Bit | Mask |
|--------------|---------------|-----|------|
| Selected     | SELECTED      | 0   | 1    |
| Crossed      | CROSSED       | 1   | 2    |
| Checked      | CHECKED       | 2   | 4    |
| Disabled     | DISABLED      | 3   | 8    |
| Outlined     | OUTLINED      | 4   | 16   |
| Shadowed     | SHADOWED      | 5   | 32   |

The GEMAES.BH defines these items as masks, pre-facing them with `mask_`. There are HGT routines that are analogous to those for object flags. So `Curob_state(i,mask_selected)` returns true if the *i*th object is selected, `Inclob_state i,mask_disabled` disables object *i* and `Exclob_state`

## Border Thickness

The low byte of the high word in some `ob_spec` fields stores the border thickness in pixels. A value of 0 means no border, positive values give a border inside the object, negative values force it outside the object.

## Colour Word

The colour word used in some `ob_spec` fields consists of the following components:

| Border Colour |    | Text Colour |   | X/O | Fill Pattern |   | Fill Colour |   |
|---------------|----|-------------|---|-----|--------------|---|-------------|---|
| 15            | 12 | 11          | 8 | 7   | 6            | 4 | 3           | 0 |

In the above diagram the numbers indicate the bits, so that the Border Colour is in bits 15-12, the four most significant bits of the first byte.

X/O is the Transparent/Opaque bit; Opaque is indicated by the bit being set.

Fill Pattern is as on the Fill menu with 0 indicating hollow- and 7 solid-fill.

The Border, Text and Fill Colours are as on the appropriate menus. The standard names for the colours are:

| Colour  | Value | Colour   | Value |
|---------|-------|----------|-------|
| WHITE   | 0     | LWHITE   | 8     |
| BLACK   | 1     | LBLACK   | 9     |
| RED     | 2     | LRED     | 10    |
| GREEN   | 3     | LGREEN   | 11    |
| BLUE    | 4     | LBLUE    | 12    |
| CYAN    | 5     | LCYAN    | 13    |
| YELLOW  | 6     | LYELLOW  | 14    |
| MAGENTA | 7     | LMAGENTA | 15    |

The L in the above names indicates *light*. If you *must* encode a colour word into your program the best base to use is hexadecimal.

### TEDINFO Structure

This structure is used by the object types BoxText, FBoxText, FText and Text:

|              |          |  |
|--------------|----------|--|
| te_ptext     | longword | pointer to actual text   |
| te_ptmplf    | longword | pointer to template; editable portion denoted by underscores                   |
| te_pvalid    | longword | pointer to string containing validation characters                             |
| te_font      | word     | font used: 3=system font, 5=small font   |
| te_resvd1    | word     | reserved for future use  |
| te_just      | word     | text justification required: 0=left, 1=right, 2=centre                         |
| te_color     | word     | object colour and pattern of box-type objects (see previously for word format) |
| te_resvd2    | word     | reserved for future use  |
| te_thickness | word     | border thickness   |
| te_txtlen    | word     | length of te_ptext string (including null)                                     |
| te_tmplen    | word     | length of te_ptmplf string (including null)                                    |

The HGT defines Sette\_ptext and Gette\_ptext\$ to set and return BASIC strings from/to the te\_ptext field. For example, Sette\_ptext i,newted\$ will set the te\_ptext field of the ith object to be newted\$. Similarly Gette\_ptext(i) will return the text field corresponding to the ith object.

### ICONBLK Structure

This is used by the Icon object type only:

|          |          |   |
|----------|----------|---|
| ib_pmask | longword | pointer to icon mask  |
| ib_pdata | longword | pointer to icon data  |
| ib_ptext | longword | pointer to the text displayed with the icon   |
| ib_char  | word     | low byte is the displayed character, high byte defines colour used - top nibble is foreground colour, bottom nibble is background |
| ib_xchar | word     | X co-ordinate of ib_char relative to ib_xicon   |
| ib_ychar | word     | Y co-ordinate of ib_char relative to ib_yicon   |
| ib_xicon | word     | X co-ordinate of icon relative to the ob_x of the object  |
| ib_yicon | word     | Y co-ordinate of icon relative to the ob_y of the object  |
| ib_wicon | word     | width of the icon image in pixels (must be a multiple of 16)  |
| ib_hicon | word     | height of icon image in pixels  |
| ib_xtext | word     | X co-ordinate of icon's text relative to the ob_x of the object   |
| ib_ytext | word     | Y co-ordinate of icon's text relative to the ob_y of the object   |
| ib_wtext | word     | width of rectangle to display icon's text in (centred)  |
| ib_htext | word     | height of icon's text   |

The bit images for the mask and data are stored as arrays of words.

## **BITBLK Structure**

This is used by the Image object type and Free Images only:

|          |          |   |
|----------|----------|---|
| bi_pdata | longword | pointer to bit image                        |
| bi_wb    | word     | width of Image data in bytes (must be even) |
| bi_hl    | word     | height of Image in pixels                   |
| bi_x     | word     | source X co-ordinate                        |
| bi_y     | word     | source Y co-ordinate                        |
| bi_color | word     | colour word (see previously)                |

bi\_x and bi\_y are used as offsets into the bit image given by bi\_pdata; any bits before this will be ignored.

## **APPLBLK Structure**

This is used by ProgDefs:

|         |          |  |
|---------|----------|--|
| ab_code | longword | pointer to code to draw the object                     |
| ab_parm | longword | passed as a parameter to the low-level drawing routine |

## **PARMBLK Structure**

This is passed to ProgDef drawing routines:

|              |          |   |
|--------------|----------|---|
| pb_tree      | longword | pointer to start of object tree                                     |
| pb_obj       | word     | the object index  |
| pb_prevstate | word     | the old state of the object to be changed                           |
| pb_currstate | word     | the new (changed) state of the object                               |
| pb_x, pb_y   | words    | the pixel X and Y screen co-ordinates of the object                 |
| pb_w, pb_h   | words    | the pixel width and height of the object                            |
| pb_xc, pb_yc | words    | the pixel X and Y screen co-ordinates of the current clip rectangle |
| pb_wc, pb_hc | words    | the pixel width and height of the current clip rectangle            |
| pb_parm      | longword | copied from the address after the pointer to the drawing routine    |

If pb\_prevstate and pb\_currstate are the same then the AES is drawing the object, not changing it.

## **Hints & Tips on Resources**

### **Using ProgDefs**

If a loaded resource file contains any ProgDef objects, their ob\_spec field will not be initialised on loading - this is up to the programmer. An APPLBLK structure needs to be allocated and initialised, then a pointer to it planted in the relevant ob\_spec field.

The drawing routine (planted in the ab\_code field) will then be called whenever that object needs drawing or changing. The routine should normally be written in assembly-language, save all registers except D0 and must not call the AES under any circumstances as the AES is *not* re-entrant.

When your custom drawing routine has finished the value it returns is the ob\_state value which you wish the AES to render over your object, i.e. returning a value of 0 applies no extra effects, whereas returning CROSSED (for instance) would draw a cross over the object. Note that any number of ob\_state values may be ORed together to produce the required effect..

When designing ProgDefs it is often easiest to base them on existing objects which can be manipulated in WERCS, e.g. in the example program we implement a rounded button based on the normal square button, hence the text may be manipulated from within WERCS.

When the AES calls your drawing code you are still in the AES' 'context', i.e. you are using it's stack, hence recursive routines or large local arrays may cause it to overflow.

If you draw using the AES' handle then you should ensure that you maintain any of the VDI attributes, alternatively you may use your own virtual workstation which will avoid these problems.

### **Creating New Desktops**

It is possible to replace the standard GEM background pattern (the area of the screen not used by the menu bar) using a special tree. This allows different colours and fill patterns to be used, as well as allowing icons to appear on the desktop.

A Form should be created in WERCS with the root object being a borderless Box with a suitable fill pattern and colour. If any icons are required these should be added to this Form. The size of the Form is not relevant. To tell GEM to use this Form, the size and position (ob\_x, ob\_y, ob\_width and ob\_height) fields in the root object should be set to the usable screen size, found using the AES wind\_get(DESK, WF\_WORKXYWH, ... call. The form can then be installed using a wind\_set(WF\_NEWDESK, ... call with an object parameter of zero. Before your program terminates, the desktop must be de-installed by passing a NULL value to the same call.

Note that installing a desktop does not cause it to be drawn and you should normally call form\_do(FMD\_FINISH, ... to force a redraw of the area.

The HiSoft GEM Toolbox NEWDESK.BAS file shows this in practice.

## Common Mistakes and how to avoid them

The following is a list of common mistakes made when programming with GEM and resources in general. The reasons given here are brief as there is insufficient space to expand upon them; they act as pointers for where to look in other documentation.

**Problem:** My program was mainly working but now it crashes during its initialisation.

**Reason:** Your resource file is out of step with your program and what was the Menu that you were displaying is now a Form; as a result the GEM menu\_bar call bombs. Re-compile all the parts of your program that rely on the header file.

**Problem:** My dialog box doesn't disappear after you click on OK.

**Reason:** The dialog box is on top of one of your windows and you are not replying to WM\_REDRAW events. If you don't open a window, the Desktop will re-draw the desktop tree for you automatically.

**Problem:** My program crashes when it should be displaying a Dialog Box.

**Reason 1:** If you have no editable fields and are passing -1 as the starting object the machine may crash, despite what some documentation says. Use 0 instead.

**Reason 2:** If you do have editable text fields make sure that they conform to the rules under FBoxText regarding editable text.

**Problem:** A GEM program crashes unexpectedly. After rebooting, the same program works correctly under the same conditions.

**Reason:** A program has modified GEM's data structures unintentionally. There are many possible ways of doing this; one to look out for is not doing a v\_clsvwk after a v\_opnvwk; that is leaving a Virtual Workstation open.

**Problem:** The mouse disappears or leaves extra pixels on the screen ('mouse droppings').

**Reason:** Your graf\_mouse calls are mis-balanced in some way. For each hide (M\_ON) call you *must* have a show (M\_OFF) call.

**Problem:** There are mouse droppings where a menu has been pulled down.

**Reason:** You are not using wind\_update (BEG\_UPDATE, ... and graf\_mouse (or the VDI v\_hide\_c) before writing to the screen.

**Problem:** When using some desk accessories the screen display is messed up.

**Reason:** Make sure that you are taking note of WM\_REDRAW events and only updating the areas given by the wind\_get (WF\_FIRSTXYWH, ... and wind\_get (WF\_NEXTXYWH, ... calls. To test this, move a desk accessory about the screen; the Control Panel and the Saved! desk accessory can both be used.

**Problem:** Some desk accessories 'lose' their mouse when invoked from my program.

**Reason:** Make sure you don't remove the mouse until *after* you have done a wind\_update (BEG\_UPDATE, ... call and make sure that it is visible before calling wind\_update for END\_UPDATE.

**Problem:** The program works fine in medium and high resolution, but crashes when accessing a menu on 'old' ROM machine.

**Reason:** Your menu is taking up more than one quarter of the screen. When running in Low Resolution, a menu may not contain more than 16000 pixels. If you are using large menus, you may wish to consider using a special menu for low resolution, as WERCS does.

## **.BH file details**

The .BH file produced by WERCS consists of constant declarations of the form:

```
CONST label%=1
```

If you have a saved a resource file called TEST.RSC you would then include the constants from the name file using:

```
rem $include test.bh
```

The characters allowed in names are:

A-Z, a-z as the first character and:

A-Z, a-z, 0-9, \_ and . in subsequent characters.

## **The WTEST Example Program**

To illustrate the most common resource-handling programming requirements, we supply an example program called WTEST.BAS. A ready-to-run version called WTEST.PRG is supplied on the master disks; it needs WRSC.RSC to run.

When compiling the source code version you will need the GEMAES.BH file from your HiSoft BASIC master disk as well as the WRSC.BH file produced by WERCS.

## **WTEST structure**

The program is deliberately over-simplified; it manages to avoid calling the VDI completely and gets away with an `evnt_mesag`, avoiding the dreaded `evnt_multi`. The general structure of the code in all the programs is as follows:

### **Procedure INITIALISE**

This does the required GEM initialisation then loads the resource file. The tree address of the menu is found and the menu installed. The usable screen size is found and certain global variables initialised.

### **Procedure SETDESK**

This sets the new desktop pattern to be a particular address and forces the AES to re-draw the whole screen.

### **Procedure DEINITIALISE**

Resets any installed desktop, removes the menu bar, frees the resource, then does any required GEM de-initialisation.

### **Procedure HANDLE\_DIALOG**

A general dialog box handler which starts by centring and drawing the box. User interaction is handled by `form_do` and, on return, if the exit object was a Button, it is de-selected.

### **Procedure SET\_TEDINFO**

This allows a particular TEDINFO structure to have its data portion set to a particular string.

### **Procedure GET\_TEDINFO**

Allows a particular TEDINFO structure to return its data portion.

### **Procedure SET\_BUTTON**

This allows one particular radio Button to be set from a group. If invalid parameters are specified the routine will never finish.

### **Procedure GET\_BUTTON**

Allows a group of radio Buttons to be interrogated to see which is selected.

### **Procedure TEST\_DIALOG**

This handles the particular dialog box in the resource file. It implements proper cancelling - that is, if it is cancelled, the Button state and Text entry are left alone.

### **Procedure HANDLE\_MENU**

This is the menu-click dispatcher; it takes various actions, depending on which menu item has been clicked, and also de-selects the menu title.

### **Procedure MAIN**

This is the main loop, acting only on `MN_SELECTED` message events. In a proper program `evnt_multi` would be used and a far greater selection of cases would have to be dealt with.



# Chapter 5

## MonSTB Reference

---

This is the reference manual for the debugger that is supplied with HiSoft BASIC. Note that on the TT you must use MonTTB.

For details on how to use the debugger with BASIC programs, please refer to *Chapter 3* of the *User Manual*.

### Numeric Expressions

---

MonSTB has a full expression evaluator, based on that in the DevpacST assembler, GenST, including operator precedence.

The following operators are supported, in decreasing order of precedence:

- monadic minus (-) and plus (+), address of line number (#)
- bitwise not (~)
- shift left (<<) and shift right (>>)
- bitwise And (&), Or (!) and Xor (^)
- multiply (\*) and divide (/)
- addition (+) and subtraction (-)
- equality (=), less than (<), greater than (>), not equals (<> or !=), less than or equal (<=), greater than or equal (>=)

The comparison operators are signed and return 0 if false or -1 (\$FFFFFFF) if true. The shift operators take the left hand operand and shift it the number of bits specified in the right hand operand, vacated bits are filled with zeroes.

This precedence can be overridden by the use of parentheses ( and ). With operators of equal precedence, expressions are evaluated from left-to-right. Spaces in expressions (other than those within quotes - ASCII constant\$) are not allowed.

All expression evaluation is done using 32-bit signed-integer arithmetic, with no checking of overflow.



The MonSTB expression evaluator also supports indirection using the { and } symbols. Indirection may be performed on a byte, word or long basis, by following the } with a period then the required size, which defaults to long. If the pointer is invalid, either because the memory is unreadable or not an even address (if word or longword indirection is used) then the expression will not be valid.

For example, the expression

```
{data_start+10}.w
```

will return the word contents of location `data_start+10`, assuming `data_start` is even. Indirection may be nested in a similar way to ordinary parentheses.

## Numbers

Absolute numbers may be in various forms:

decimal constants, e.g. `\1029`

hexadecimal constants, e.g. `12f` or `$12f`

octal constants, e.g. `@730`

binary constants, e.g. `%1100010`

character constants, e.g. `'X'`

`\` is used to denote decimal numbers, `$` is used to denote hexadecimal numbers (the default), `%` for binary numbers, `@` for octal numbers and single `'` or double `"` quotes for character constants.

## Character Constants

Whichever quote is used to mark the start of a string must also be used to denote its end and quotes themselves may be used in strings delimited with the same quote character by having it occur twice. Character constants can be up to 4 characters in length and evaluate to right-justified longs with null-padding if required. For example, here are some character constants and their ASCII and hex values:

|                     |                   |                         |
|---------------------|-------------------|-------------------------|
| <code>"Q"</code>    | <code>Q</code>    | <code>\$00000051</code> |
| <code>'hi'</code>   | <code>hi</code>   | <code>\$00006869</code> |
| <code>"Test"</code> | <code>test</code> | <code>\$54657374</code> |
| <code>"it's"</code> | <code>it's</code> | <code>\$6974277C</code> |
| <code>'it's'</code> | <code>it's</code> | <code>\$6974277C</code> |

## Symbols and Registers

Symbols may be referred to and are normally case-sensitive and significant to either 8 or 22 characters (depending on the form of debug information used), though this can be changed.

Registers may be referred to simply by name, such as `A3` or `D7` (case insensitive), but this clashes with hex numbers. To obtain such hex numbers precede them with either a leading zero or a `$` sign. `A7` refers to the *user* stack pointer.

There are several reserved symbols which are case insensitive, namely `TEXT`, `DATA`, `BSS`, `END`, `SP`, `SR`, and `SSP`. `END` refers to one byte past the end of the `BSS` section and `SP` refers to either the user- or supervisor-stack, depending on the current value of the status register. Remember that the names of all your external variables and functions will also be available.



In addition you can access the `SFC`, `DFC`, `CACR`, `CAAR`, `VBR`, `MSP`, `ISP`, `MMUSR`, `TT0`, `TT1`, `TC`, `FPCR`, `FPSR`, and `FPIAR` registers.

Note that the `CRP`, `SRP` cannot be used in expressions since they are 64 bits long.

In addition there are 10 memories numbered `M0` through `M9`, which are treated in a similar way to registers and can be assigned to using the `Register Set` command. Memories 2 through 5 inclusive refer to the current start address of the relevant window and modifying them will change the start address of that window.

## Window Types

There are four window types and the exact contents of these windows and how they are displayed is detailed below. The allowed types of windows are shown in the table below.

| Window | Allowed Types                      |
|--------|------------------------------------|
| 1      | Register                           |
| 2      | Disassembly                        |
| 3      | Memory                             |
| 4      | Disassembly, Memory or Source code |
| 5      | Memory                             |

## Register Window Display

The data registers are shown in hex, together with the ASCII display of their low byte and then a hex display of the eight bytes they point to in memory. The address registers are also shown in hex, together with a hex display of 12 bytes. As with all hex displays in MonSTB this is word-aligned, with non-readable memory displayed as \*\*.

The status register is shown in hex and in flag form, additionally with U or S denoting user- or supervisor-modes. A7' denotes the supervisor stack pointer, displayed in a similar way to the other address registers.

The PC value is shown together with a disassembly of the current instruction. Where this involves one or more effective addresses these are shown in hex, together with a suitably-sized display of the memory they point to.

For example, the display

```
TST.W $12A(A3) ;00001FAE 0F01
```

signifies that the value of \$12A plus register A3 is \$1FAE, and that the word memory pointed to by this is \$0F01. A more complex example is the display

```
MOVE.W $12A(A3), -(SP) ;00001FAE 0F01 ⇒0002AC08 FFFF
```

The source addressing mode is as before but the destination address is \$2AC08, presently containing \$FFFF. Note that this display is always of a suitable size and when pre-decrement addressing is used this is included in the address calculations.



No hex data is shown for the data registers and the address register data area is reduced to 4 bytes. In addition the disassembly line may not be long enough to display complex addressing modes such as the second example above.



The floating point registers (if displayed) are shown first in internal hexadecimal format (omitting the reserved 0 word to save screen space) and in decimal scientific notation. Infinities and NaNs (not a number) are shown appropriately. Note that the decimal value shown does not give the full accuracy expressible in extended format; however the hexadecimal value is precise - you should also note that the trigonometric facilities of the maths co-processors are only accurate to double rather than extended precision.

## Disassembly Window Display

Disassembly windows display memory as disassembled instructions. On the left the hex address is shown, followed by any symbol, then the disassembly itself. The current value of the PC is denoted with ⇒.

If the instruction has a breakpoint placed on it this is shown using square brackets [ ] afterwards, the contents of which depend on the type of breakpoint. For stop breakpoints this will be the number of times left for this instruction to execute, for conditional breakpoints this will be a ? followed by the beginning of the conditional expression, for count breakpoints this will be an = sign followed by the current count, and for permanent breakpoints an \* is shown.

The exact format of the disassembled op-codes is Motorola standard, as accepted by the assembler, *asm*. All output is upper-case (except lower-case labels) and all numeric output is hex, except trap numbers. Leading zeroes are suppressed and the \$ hex delimiter is not shown on numbers less than 10. Where relevant numerics are shown signed. The only deviation from Motorola standard is the register lists shown in MOVEM instructions - in order to save display space the type of the second register in a range is abbreviated, for example

```
MOVEM.L d0-d3/a0-a2, -(sp)
```

will be disassembled as

```
MOVEM.L d0-3/a0-2, -(sp)
```



Any displayed symbols replace the hex address display, limited to a maximum of 8 characters.

## Memory Window Display

Memory windows display memory in the form of a hex address, word-aligned hex display and ASCII. Unreadable memory locations are denoted by \*\*. The number of bytes shown is calculated from the window width, up to a maximum of 16 bytes per line.

## Source-code Window Display

The source-code window displays ASCII files in a similar way to a screen editor. The default tab setting is 8 though this can be toggled to 4 with the *Edit Window* command.

## Window Commands

The Alt key is generally used for controlling windows, and when used applies to the *current window*. This is denoted by having an inverse title and can be changed by pressing Tab, or Alt and the window number.

Most window commands work in any window, zoomed or not, though when it does not make sense to do something the command is ignored.

### Alt-A Set Address

This sets the starting address of a memory or disassembly window.

### Alt-B Set Breakpoint

Allows the setting of any type of breakpoint, described later under *Breakpoints*.

### Alt-E Edit Window

On a memory window this lets you edit memory in hex or ASCII. Hex editing can be accomplished using keys 1-9, A-F, together with the cursor keys. Pressing Tab switches between hex & ASCII, ASCII editing takes each keypress and writes it to memory. The cursor keys can be used to move about memory. To leave edit mode press the Esc key.

On a register window this is the same as Alt-R, *Register Set*, described shortly.

On a source-code window this toggles the tab setting between 4 and 8.

### Alt-F Font size

This changes the font size in a window. In high resolution 16 and 8 pixel high fonts are used, in colour 8 and 6 pixel high fonts are used. This allows a greater number of lines to be displayed, assuming your monitor can cope.

Changing the font size on the register window causes the position of windows 2 and 3 to be re-calculated to fill the available space.

### Alt-L Lock Window

This allows disassembly and memory windows to be locked to a particular register. After any exception the start address of the window is re-calculated, according to the locked register.

To unlock a window simply enter a blank string.

By default window 2 is locked to the PC. You can lock windows to each other by specifying a lock to a memory window, such as M2.

### Alt-O Show Other

This prompts for an expression and displays it in hex, decimal and as a symbol if relevant.

### Alt-P Printer Dump

Dumps the current window contents onto the printer. The print can be aborted by pressing Esc.

### Alt-R Register Set

Allows any register to be set to a value, by specifying the register, an equals sign, then its new value. It can also be used to set the values of the MonSTB memories M0 to M9. For example the line

a3=a2+4

sets register A3 to be A2 plus 4. You can also use this to set the start address of windows when in zoom mode so that on exit from zoom mode the relevant window starts at the required address.



This command may be used to set all the 68030 and 68881 control registers, so use it with care! The standard floating point registers (FP0-FP7) must be followed by a floating point constant either in decimal or hexadecimal using the same rules as for GenTT. Hexadecimal floating point numbers should be prefixed with either \$ or : and must use the full 12 byte extended format complete with unused second word of 0000.

This command may be used to set the 64-bit CRP and SRP registers; these must be assigned a single 16 digit hexadecimal number without any lead-in character..

## Alt-S Split windows

This either splits window 2 into window 2 and window 4, or splits window 3 into window 3 and window 5. Each new window is independent from its creator. Pressing Alt-S again will unsplit the window.



This command has no effect.

## Alt-T Change Type

This only works on window 4 (created either by splitting window 2 or by loading a source file). It changes the type of the window between disassembly, memory and source-code (if a file has been loaded).



In TT medium resolution this command may also be used on window 1 (the register window) to toggle the display of the extra 68030 and maths co-processor registers if the smaller font size has been selected.

## Alt-Z Zoom Window

This zooms the current window to be full size. Other Alt commands are still available and normal size can be achieved by pressing Esc or Alt-Z again.



Zooming the register window can be extremely useful as it allows you to see the 'hidden' registers M0, M1 and M6-M9.

## Cursor Keys

The cursor keys can be used on the current window, and their action depends on the window type.

On a memory window all four cursor keys change the current address, and Shift ↑ and Shift ↓ move a page in either direction.

On a disassembly window ↑ and ↓ change the start address on an instruction basis, ← and → change the address on a word basis.

On a source-code window ↑ and ↓ change the display on a line basis, and Shift ↑ and Shift ↓ on a page basis.

## Screen Switching

MonSTB uses its own screen display and drivers to prevent interference with a program's own screen output. To prevent flicker caused by excessive screen switching when single-stepping the screen display is only switched to the program's after 20 milliseconds, producing a flicker-free display while in the debugger. In addition the debugger display can have a different screen resolution to your program's if using a colour monitor.

## V View Other Screen

This flips the screen to that of the programs, any key returns to the MonSTB display.

## Ctrl-O Other Screen Mode

This changes the screen mode of MonST's display between low and medium resolution. It re-initialises window font sizes and window positions to that of the initial display. This will not effect the screen mode of the program being debugged.

This command is ignored on a monochrome monitor.



This cycles the screen mode of MonTT's display between the five different screen modes available on the TT when using a colour monitor.

As MonSTB has its own idea of where the screen is, what mode it is in and what palette to use you can use MonSTB to actually look at the screen memory in use by your program, ideal for low-level graphics programs.



If your program changes screen position or resolution, via the XBIOS or the hardware registers, it is important that you temporarily disable screen switching using Preferences while executing such code; otherwise MonSTB will not notice the new attributes of your program's screen.

When a disk is accessed, when loading or saving, the screen display will probably switch to the program's during the operation. This is in case a disk error occurs, such as a write-protection violation or a read error, as it allows any GEM alert boxes to be seen and acted upon.



# Breaking into Programs

## Shift-Alt-Help Interrupt Program

While a program is running it can be interrupted by pressing this key combination, which will cause a trace exception at the current value of the PC. With computationally intensive program sections this will be within the program itself but with a program making extensive use of the ROM, such as the GEMDOS or AES, the interruption will normally be in the ROM itself, or the line-F handler stored in low-memory. If this is the case it is recommended that a breakpoint be placed in your actual program area then a Return to Program command (Ctrl-R) issued.

Pressing Alt-Help without the Shift key will normally produce a screen dump to the printer - if you press this accidentally it should be pressed again to cancel the dump.

It is possible for this key combination to be ignored when pressed - if this occurs press it again and it should work. Pressing it when in MonSTB itself will produce no effect.



A program should never be terminated (using Ctrl-C) if it has just been interrupted in the middle of a ROM routine. This is likely to cause a system crash.

## Breakpoints

Breakpoints allow you to stop the execution of your program at specified points within it. MonSTB allows up to eight simultaneous breakpoints, each of which may be one of five types. When a breakpoint is hit MonSTB is entered and then decides whether or not to halt execution of your program, entering the front panel display, or continue; this decision is based on the type of the breakpoint and the state of your program's variables.

### Simple Breakpoints

These are one-off breakpoints which, when executed, are cleared and cause MonSTB to be entered.

## Stop Breakpoints

These are breakpoints that cause program execution to stop after a particular instruction has been executed a certain number of times. In fact a simple breakpoint is really a stop breakpoint with a count of one.

## Count Breakpoints

Merely counters; each time such a breakpoint is reached a counter associated with it is incremented, and the program will resume.

## Permanent Breakpoints

These are similar to simple breakpoints except that they are never cleared - every time execution reaches a permanent breakpoint MonSTB will be entered.

## Conditional Breakpoints

The most powerful type of breakpoint which allow program execution to stop at a particular address only if an arbitrarily complex set of conditions apply. Each conditional breakpoint has associated with it an expression (conforming to the rules already described). Every time the breakpoint is reached this expression is evaluated, and if it is non-zero (i.e. true) then the program will be stopped, otherwise it will resume.

## Alt-B Set Breakpoint

This is a window command allowing the setting or clearing of breakpoints at any time. The line entered should be one of the following forms, depending on the type of breakpoint required:

`<address>`

will set a simple breakpoint.

`<address>,<expression>`

will set a stop breakpoint at the given address, after it has executed `<expression>` times.

`<address>,<=>`

will set a count breakpoint. The initial value of the count will be zero.

`<address>,*`

will set a permanent breakpoint.

<address>,?<expression>

will set a conditional breakpoint, using the given expression.

<address>,-

will clear any breakpoint at the given address.

Breakpoints cannot be set on addresses which are odd or unreadable, or in ROM, though ROM breakpoints may be emulated using the Run Until command.

Every time a breakpoint is reached, regardless of whether the program is interrupted or resumed, the program state is remembered in the History buffer, described later.

## Help Show Help and Breakpoints

This displays the text, data and BSS segment addresses and lengths, together with every current breakpoint. Alt-commands are available within this display.

## Ctrl-B Set Breakpoint

This sets a simple breakpoint at the start address of the current window, so long as it is a disassembly window. If a breakpoint is already there then it will be cleared.

## U Go Until

This prompts for an address, at which a simple breakpoint will be placed then program execution resumed.

## Ctrl-K Kill Breakpoints

This clears all set breakpoints.

## Ctrl-A Set Breakpoint then Execute

A command that places a simple breakpoint at the instruction *after* that at the PC and resumes execution from the PC. This is particularly useful for DBF-type loops if you don't want to go through the loop, but just want to see the result after the loop is over.

## Ctrl-D GEMDOS Breakpoint

This allows a breakpoint to be set on specific GEMDOS calls. The required GEMDOS number should be entered, or a blank line entered if any existing GEMDOS breakpoint needs to be cleared.

## History

MonSTB has a *history buffer* in which the machine status is remembered for later investigation.

The most common way of entering data into the history buffer is by using you single-step but, in addition, every breakpoint reached and every exception caused enters the machine state into the buffer. Various forms of the Run command also cause entries to be made into this buffer.



The history buffer has room for five entries - when it fills up, the oldest entry is removed to make room for the newest entry.

## H Show History Buffer

This opens a large window displaying the contents of the history buffer. All register values are shown including the PC as well as a disassembly of the next instruction to be executed.



If a disassembly in the History display includes an instruction which has a breakpoint placed on it the [ ]s will show the *current* values for that breakpoint, not the values at the time of the entry into the history buffer.

## Quitting MonSTB

## Ctrl-C Terminate

This will issue a terminate trap to the *current* GEMDOS task. If a program has been loaded from within MonSTB it will be terminated and the message Program Terminated appear in the lower window. Another program can then be loaded, if required.



If no program has been loaded into MonSTB it will itself terminate when this command is used.



Terminating some GEM programs prematurely, before they have closed workstations or restored window control properly can seriously confuse the AES and VDI. This may not be noticeable immediately but often causes crashes when a subsequent program is executed.

## Loading & Saving

### Ctrl-L Load Executable Program

This will prompt for an executable filename then a command line and will attempt to load the file ready for execution. If MonSTB has already loaded a program it is not possible to load another until the former has terminated.

The file to be loaded must be an executable file - attempting to load a non-executable file will normally result in TOS error 66 and further attempts to load executable files will normally fail as GEMDOS does not de-allocate the memory it allocated before trying to load the errant file. If this occurs terminate MonSTB, re-execute it and use the Load Binary File command.

### B Load Binary File

This will prompt for a filename and optional load address (separated by a comma) and will then load the file where specified. If no load address is given, memory will be allocated from GEMDOS and used. M0 will be set to the start address and M1 to the end address.

### S Save Binary File

This will prompt for a filename, a start address and an (inclusive) end address. To re-save a file recently loaded with the Load Binary File command <filename>,M0, and M1 may be specified, assuming of course that M0 and M1 have not been re-assigned.

## A Load ASCII File

This powerful command allows an ASCII file, normally of source code, to be loaded and viewed within MonSTB. Window 4 will be created if required then set up as a source-code window. Memory for the source code is taken from GEMDOS so sufficient free memory must be available. It is recommended that source-code be loaded *before* an executable program to ensure enough memory.

If the file currently being debugged contains HiSoft BASIC debug information, MonSTB will use the line number information corresponding to the new source file that has been loaded. Thus, loading the source file will change the effect of the # operator.



Window 4 is not shown, though an ASCII file may be loaded in low-res then viewed after switching to medium resolution using Ctrl-O and pressing Alt-S, Alt-T, Alt-T.



If an ASCII file is loaded *after* an executable program the memory used will be owned by the *program itself*, not MonSTB. When such a program terminates, any displayed source-code window will be closed. This is also the case when the source text is automatically loaded by MonSTB.

## Executing Programs

### Ctrl-R Return to program / Run

This runs the current program with the given register values at full speed and is the normal way to resume execution after entry via a breakpoint.

### Ctrl-Z Single-Step

This single-steps the instruction at the PC with the current register values. Single-stepping a TRAP, Line-A or Line-F opcode will, by default, be treated as a single instruction. This can be changed using Preferences.

## Ctrl-Y Single-Step

Identical to Ctrl-Z above but included for the convenience of German users.

## Ctrl-T Interpret an Instruction (Trace)

This interprets the instruction at the PC using the displayed register values. It is similar to Ctrl-Z but skips *over* BSRs, JSRs, TRAPs, Line-A and Line-F calls, re-entering the debugger on return from them to save stepping all the way through the routine or trap. It works on instructions in ROM or RAM.

## Ctrl-S Skip an Instruction

Ctrl-S increments the PC register by the size of the current instruction thus causing it to be skipped. Use this instead of Ctrl-Z when you know that this instruction is going to do something it shouldn't or that you don't like.

## R Run (various)

This is a general Run command and prompts for the type of execution, selected by pressing a particular key.

### Run G Go

This is identical to Ctrl-R, Run, and resumes the program at full speed.

### Run S Slowly

This will run the program at reduced speed, remembering every step in the history buffer.

### Run I Instruction

This is similar to Run Slowly but allows a count to be entered, so that a particular number of instructions may be executed before MonSTB is entered.

### Run U Until

You will be prompted for an expression which will be evaluated after every instruction. The program will then run, albeit at reduced speed, until the given expression evaluates to non-zero (true) when MonSTB will be entered. For example if single-stepping a DBF loop which used d6 in the ROM code you could say Run Until d6&ffff=ffff (waiting for the low word of d6 to be \$FFFF) or, alternatively, PC=FC8B1A, or whatever.



This should not be confused with the Until command, which takes an address, places a breakpoint there then resumes execution.

With all of these commands (except Run Go) you will then be asked Watch Y/N? If Y is selected then the MonSTB display will be shown after every instruction and you can watch registers and memory as they change, or interrupt execution by pressing both Shift keys simultaneously. If N is selected then execution will occur while showing your program's display and execution may be interrupted by pressing Shift-Alt-Help.



Selecting Watch mode with screen switching turned off is likely to result in a great deal of eye strain as the display will be flipped after each and every instruction, particularly alarming with colour monitors.

With any of these Run modes (except Go) all information after every instruction will be remembered in the history buffer. In addition TRAPs will be treated as single-instructions, unless changed with Preferences; though see the warnings under that command about tracing all the way through ROM routines.

When a program is running with one of the above modes a couple of pixels near the top left of the display will flicker, to denote that something is happening, as it is possible to think the machine has hung when, in fact, it is simply taking a while to Run through the code, an instruction at a time.

## Searching Memory

### G search memory (Get a sequence)

This will prompt Search for B/W/L/T/I?, standing for Bytes, Words, Longs, Text and Instructions.

If you select B, W or L you will then be prompted to enter the sequence of numbers you wish to search for, each separated by commas. MonSTB is not normally fussy about word-alignment when searching, so it can find longs on odd boundaries, for example. If you wish to force a particular alignment, finish the list of items to search for with 'W'; for word boundaries or 'L' for longword boundaries.

If you select T you may search for any given text string, for which you will be prompted. You will also be asked whether you wish the search to be case sensitive; if you press Y then Test will match TEST or TeSt

If you select I you can search for part or all of the mnemonic of an instruction; for example if you searched for \$14(A you would find an instruction like MOVE.L D2,\$14(A0). The case of the string you enter *is* important (unlike MonST version 1), but you should bear in mind the format the disassembler produces, e.g. always use hex numbers, refer to A7 rather than SP and so on.

Once you have selected the search type and parameters, the search begins, control passing to the Next command, described below.

## **N find next**

This can be used after the G command to find subsequent occurrences of the search data. With the B, W, L and T options you will always find at least one occurrence, which will be in the buffer within MonSTB that is used for storing the sequence. With the T option you may also find a copy in the system keyboard buffer. With these options, the Esc key is tested every 64k bytes and can be used to stop the search. With the I option, which is very much slower, the Esc key is tested every 2 bytes.

The search area of memory goes from 0 to the end of RAM, then via the system ROM area and cartridge area then back to 0. MonSTB will not search the cartridge area if the environment variable NOCARTRIDGE exists.

The search will start just past the start address of the current window (except register windows) and, if an occurrence is found, it will re-display the window at the given address.

## **Searching Source-Code Windows**

If the G command is used on a source-code window the T sub-command is chosen automatically and, if the text is found, the window will re-display the line containing it.

# **Miscellaneous**

## **Ctrl-P Preferences**

This permits control over various options within MonSTB. The first three require Y/N answers, pressing Esc aborts and Return leaves them alone.

### **Screen Switching**

Defaulting to On, this causes the display to switch to that of your program only after 20 milliseconds. It should be switched off when a program is about to change a screen's address or resolution and then turned back on afterwards.

### **Follow Traps**

By default, single-stepping and the various forms of the Run command treat TRAPs, Line-A and Line-F calls as single instructions. However by turning this option On the relevant routines will be entered allowing ROM code to be investigated.

If you are interested in this sort of low-level hackery, you should consider purchasing DevpacST as it provides facilities from recovering from the after effects of interrupting the operating system code.

### **Relative Offsets**

This option defaults to On and affects the disassembly of the address register indirect with offset addressing modes, i.e. xxx(An). With the option on, the current value of the given address register is added to the offset and then searched for in the symbol table. If found it is disassembled as symbol(An). This option is required to show the addresses of your global variables if they are accessed via an address register.

### **Ignore Case**

This option defaults to Off. If it is set to On then if you enter fred in an expression the subsequent search will give the value of the first symbol that matches this, ignoring case, thus finding FRED, fred or Fred. This option is useful for lazy typists who use the same name with different casing.

## Show Line Numbers in Source

MonSTB can either show line numbers in your source window in decimal (press D), hexadecimal (H) or not at all (press N). Using hexadecimal line numbers has the advantage that you can use them directly with the # line number operator. This if you can see that you want to execute your program until the line with number 001C then just type U (for run until) #1C. Remember how ever that A0 through A7 and D0 through D7 are register names and take priority over hexadecimal numbers. To enter line number A0 use #A0.

Decimal line numbers are naturally more civilised but remember that you need to prefix any decimal number with \. If you want to find the address of line 28 decimal, use #\28 not #28.

## Auto Load Source

Using the default settings, MonSTB will automatically load a HiSoft BASIC source file and run your program, ready for you to set a breakpoint in the code. MonSTB loads the source file corresponding to the main file. rather than any include files. You can disable this feature if you do not wish to load this source file or you wish to debug a program written in another language.

## Automatic Prefix Labels

Using the default setting MonSTB will try prefixing symbols by \_ and @ if it cannot find a label, so that if you enter start and there is no label called start, the MonSTB will try \_start or if this doesn't exist then it will try @start.

This facility is extremely useful since BASIC functions often have an \_ added by the compiler. If you are using a mixed-language program, the names may begin with @. In each case, the automatic prefixing allows you to just use the name without bothering about the prefix.

Should there be an assembly language or C name the same as a BASIC name, say test and \_test, then you will need to use the \_ explicitly to get the BASIC function rather than the C or assembly language one. You can disable this option so that only exact matches of names are supported.

## Symbols Option

This allows control over the use of symbols in expressions in MonSTB. It will firstly ask whether the case of symbols should be ignored, pressing Y will cause case independent searching to be used. It will then prompt for the maximum length of symbols, which is normally 22 but may be reduced to as low as 8.

## Top Of RAM

This indicates to MonSTB which memory location should be considered the top of memory by the Search Memory (G) command. Normally you will not need to change this as it defaults to the system variable phys\_top; but you may need to modify it if you are debugging software that lowers phys\_top.



The TT version has two tops of RAM one for ST RAM and one for TT RAM.

## Save preferences

Reply Y to this command to save your current preferences to the file MONST2.INF in the current directory. When MonST2 loads it will read your current preferences from this file. MONST.INF must be in the current directory when MonSTB is loaded.



The TT version has uses MONTT.INF

## I Intelligent Copy

This copies a block of memory to another area. The addresses should be entered in the form

<start>,<inclusive\_end>,<destination>

The copy is intelligent in that the block of memory may be copied to a location which overlaps its previous location.



No checks at all are made on the validity of the move; copying to non-existent areas of memory is likely to crash MonSTB and corrupting system areas may well crash the machine.

## L List Labels

This opens up a large window and displays all loaded symbols. Any key displays the next page, pressing Esc aborts. The symbols will be displayed in the order they were found on disk.



## W Fill Memory With

This fills a section of memory with a particular byte. The range should be entered in the form

<start>,<inclusive\_end>,<fillbyte>

The warning described under the l command about the lack of checks applies equally to this command.

## M Modify Address

Included for compatibility with MonST1, equivalent to Alt-A.

## O Show Other Bases

Included for compatibility with MonST1, equivalent to Alt-O.

## D Change Drive & Directory

This allows the current drive and sub-directory to be changed.

## Ctrl-Alt-Numeric Dot      Reset machine

Holding down Ctrl and Alt and then pressing the Dot (.) key on the numeric keypad will cause the machine to be reset. Great for Mega ST owners with 1.2 ROMs but without long arms!

## Ctrl-E Re-Install Exceptions

This command causes MonSTB to re-install the exception vectors; useful if you want the divide by 0, array bounds and overflow checks to be trapped by MonST rather than as BASIC runtime errors..

This must be used *after* the user's program has modified the exceptions - i.e. when the program has reached at least the label REF0001.

## Command Summary

### Window Commands

|       |                |
|-------|----------------|
| Alt-A | Set Address    |
| Alt-B | Set Breakpoint |
| Alt-E | Edit Window    |
| Alt-F | Font Size      |
| Alt-L | Lock Window    |
| Alt-O | Show Other     |
| Alt-P | Printer Dump   |
| Alt-R | Register Set   |
| Alt-S | Split Windows  |
| Alt-T | Change Type    |
| Alt-Z | Zoom Window    |

### Screen Switching

|        |                   |
|--------|-------------------|
| V      | View Other Screen |
| Ctrl-O | Other Screen Mode |

### Breakpoints

|        |                             |
|--------|-----------------------------|
| Alt-B  | Set Breakpoint              |
| Help   | Show Help and Breakpoints   |
| Ctrl-B | Set Breakpoint              |
| U      | Go Until                    |
| Ctrl-K | Kill Breakpoints            |
| Ctrl-A | Set Breakpoint then Execute |
| Ctrl-D | GEMDOS Breakpoint           |

### Loading and Saving

|        |                         |
|--------|-------------------------|
| Ctrl-L | Load Executable Program |
| B      | Load Binary File        |
| S      | Save Binary File        |
| A      | Load ASCII File         |

## Executing Programs

---

|        |                                  |
|--------|----------------------------------|
| Ctrl-R | Return to program / Run          |
| Ctrl-Z | Single-Step                      |
| Ctrl-Y | Single-Step                      |
| Ctrl-T | Interpret an Instruction (Trace) |
| Ctrl-S | Skip Instruction                 |
| R      | Run (various)                    |

## Searching Memory

---

|   |                                |
|---|--------------------------------|
| G | Search Memory (Get a sequence) |
| N | Find Next                      |

## Miscellaneous

---

|                |                          |
|----------------|--------------------------|
| Ctrl-Alt-Dot   | Reset machine            |
| Ctrl-C         | Terminate                |
| Ctrl-E         | Re-install breakpoints   |
| Ctrl-P         | Preferences              |
| D              | Change Drive & Directory |
| H              | Show History Buffer      |
| I              | Intelligent Copy         |
| L              | List Labels              |
| M              | Modify Address           |
| O              | Show Other Bases         |
| W              | Fill Memory With         |
| Shift-Alt-Help | Interrupt Program        |

## Debugging Stratagem

---

### Hints & Tips

---

If you have interrupted a program using Shift-Alt-Help or by a Run Until command and have found yourself in the middle of the ROM, there is a way of returning to the exact point in your program which called the ROM. Firstly ensure the Follow Traps option is on, then do Run Until with an expression of `sp=a7`. This will re-enter MonSTB the moment user mode is restored which will be in your program.

When using Run Until knowing that it will take a quite a while for the condition to be satisfied, give MonSTB a hand by pre-computing as much of the expression as you can, for example

```
(a3>(3A400-\100+M1))
```

could be reduced to

```
a3>xxx
```

where xxx has been calculated by you using the Alt-O command.

If you do use a label with Run Until then explicitly including any leading \_ or @ will speed up the table search considerably.

## Bug Hunting

---

There are probably as many strategies for finding bugs as there are programmers; there is really no substitute for learning the hard way, by experience. However, here are some hints which we have learnt, the hard way!

Firstly, a very good way of finding bugs is to look at the source code and think. The disadvantage of reaching first for the debugger, then second for the source code, is that it gets you into bad habits. You may switch to a machine or programming environment that does not offer debugging, or at least not one as powerful you are used to.

If a program fails in a very detectable way, such as causing an exception, debugging is normally easier than if, say, a program sometimes doesn't quite work exactly as it should.

Many bugs are caused by a particular memory location being stepped on. Where the offending memory location is detectable, by producing a bus error, for example, then a conditional breakpoint placed at one or more main subroutines can help greatly. For example, suppose the global variable at address 123456 is inadvertently becoming negative during execution. The conditional expression could be set up as

```
{123456}<0
```

If this method fails, and the global variable is being corrupted somewhere un-detectable, the remaining solution is to Run Until that expression, which could take a considerable time. Even then it may not find it, for example if the bug is caused by an interrupt happening at a certain time when the stack is in a particular place.



Count breakpoints are a good way of tracking down bugs *before* they occur. For example, suppose a particular subroutine is known to eventually fail but you cannot see why, then you should set a count breakpoint on it, then let the program run. At the point where the program stops, because of an exception say, look at the value of the count breakpoint (using Help). Terminate the program, re-load it, then set a stop breakpoint on the subroutine for that particular value or one before it. Let it run and then you can follow through the subroutine on the very call that it fails, to try and work out why.

## Exceptions

MonSTB uses the 68000 processor exceptions to stop runaway programs and to single-step, so at this point it would be useful to explain them and what normally happens when they occur on an ST/TT.

There are various types of exception that can occur, some deliberately, others accidentally. When one does occur the processor saves some information on the SSP, goes into supervisor mode and jumps to an exception handler. BASIC programs use some of these for detecting runtime errors such as arithmetic overflow.

When MonSTB is active it re-directs some of these exceptions so it can take control when they occur. You will need to use the Ctrl-E (Re- Install exceptions ) command after your program has loaded in order for MonST to regain control of the exceptions that are used by runtime errors.

The various forms of exceptions, their usual results, and what happens when they occur with MonSTB active is shown in the following table (on the next page):

| Number | Exception           | BASIC uauge        | MonSTB active            |
|--------|---------------------|--------------------|--------------------------|
| 2      | bus error           | bombs              | trapped                  |
| 3      | address error       | bombs              | trapped                  |
| 4      | illegal instruction | bombs              | trapped                  |
| 5      | zero divide         | Runtime error      | trapped                  |
| 6      | CHK instruction     | Runtime error      | trapped                  |
| 7      | TRAPV instruction   | Runtime error      | trapped                  |
| 8      | privilege violation | bombs              | trapped                  |
| 9      | trace               | bombs              | used for single-stepping |
| 10     | line 1010 emulator  | fast VDI interface | fast VDI interface       |
| 11     | line 1111 emulator  | internal TOS       | internal TOS             |
| 32     | TRAP #0             | bombs              | trapped                  |
| 33     | TRAP #1             | GEMDOS call        | GEMDOS call              |
| 34     | TRAP #2             | AES/VDI call       | AES/VDI call             |
| 35     | TRAP #3             | bombs              | trapped                  |
| 36     | TRAP #4             | Trace Line number  | trapped                  |
| 37-44  | TRAP #5-#12         | bombs              | trapped                  |
| 45     | TRAP #13            | XBIOS call         | XBIOS call               |
| 46     | TRAP #14            | BIOS call          | BIOS call                |
| 47     | TRAP #15            | bombs              | trapped                  |

Exceptions 2 to 8 are caused by a programmer error and are trapped by MonSTB.

Exception 9 can remotely be caused by programmer error and is used by MonSTB for single stepping.

Exceptions 10, 11, 33, 34, 45 and 46 are used by the system and left alone.

The rest (i.e. the unused TRAP exceptions) are diverted into MonSTB, but can subsequently be re-defined to be exploited by programs if required.

The TRAP #4 vector is only modified if you using the TRON command,.

The 'bombs' entry in the table above means that the ST will attempt to recover from the exception, but it is not always successful.

When an exception occurs, the ST prints on the screen a number of *bomb* shapes (or *mushrooms* on the old disk-loaded TOS), the number being equal to the exception number. Having done this, it will abort the current program (losing any unsaved data from it) and attempt a return to the Desktop.

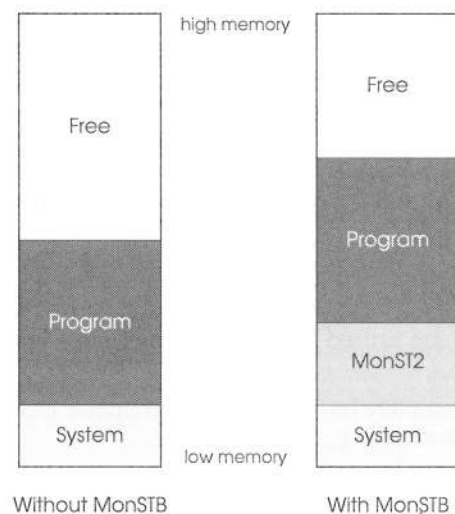
If the exception was caused by or resulted in important system variables being destroyed then the attempt may fail and the machine will not recover.

Occasionally very nasty crashes can cause the whole screen to fill with bombs (or mushrooms) which looks very impressive, but is not very useful!

## Memory Layout

The usual versions of MonSTB co-reside with programs being debugged; that is, they are loaded, ask for a filename, and load that file in together with any labels.

It is useful to examine the usual logical memory map both with and without MonSTB, shown below:



The actual code size of MonSTB is around 25k, but in addition it requires an additional 32k of workspace. This may seem large but it is required for the copy of the ST screen memory saved by MonSTB; this is a most useful feature of the debugger.

## Exception Analysis

When an unexpected exception occurs, it can be useful to be able to work out where and why it occurred and, possibly, to resume execution. Often a quick curse and a look at your source code may get your program working quicker though!

### Bus Error

If the PC is in some non-existent area of memory then look at the relevant stack to try and find a return address to give a clue as to the cause. If the PC is in a correct area of your program then the bus error must have been caused by a memory access to non-existent or protected memory. Recovering from bus errors and resuming execution is generally not possible.

### Address Error

If the PC is somewhere strange the method above should be used, otherwise the error must have been caused by a program access to an odd address. Correcting a register value may be enough to resume execution, at least temporarily.

### Illegal Instruction

If the PC is in very low memory, below around \$30, it is probable that it was caused by a jump to location 0. If you use MonSTB to look here you will see a short branch together with, normally, various ORI instructions (really longword pointers) and eventually an illegal instruction.

### Privilege Violation

This is caused by executing a privileged instruction in user mode, normally meaning your program has gone horribly wrong. Bumping the PC past the offending instruction is unlikely to be much help in resuming the program.

## Using MonSTB with other languages

---

A major feature of MonSTB is its ability to use symbols taken from the original program whilst debugging. MonSTB supports two formats for label information - the DRI standard, which allows up to 8 characters per symbol, and the HiSoft Extended Debug format, allowing up to 22 characters.

Most of HiSoft's language products support both formats (for example, DevpacST, Lattice C 5 and FTL Modula-2) and many other vendors' compilers and linkers have an option to produce DRI-format debugging information.

The line number information format is, at the present time, specific to HiSoft BASIC although MonSTB also reads the line number information output by Lattice C.

## Using MonSTB with multi-module programs

---

MonSTB will initially read the line number information and source file for the main BASIC program assuming that you have switched on line number information globally. If you wish to debug more than one file at once, then you can switch to another file by explicitly loading the appropriate source file using the A command.

However if you are only interested in debugging one include file, you can enable line numbers in an include using `rem $option &+` and then disable them using `rem $option &-` within that same include file. The include source file will then be loaded first.

## For Devpac MonST2 Users

---

If you are used to the version of MonST that is part of Devpac ST version 2, here are the differences:

- Source line numbers can now be displayed in either hexadecimal or decimal. This is set using the Preferences command.
- Using the default settings, MonSTB will automatically run the program until the label REF0001 and load the source file corresponding to the first debug information in the file.

- The operator # is used to give the address corresponding to a given BASIC line number. To use this you need to use the compiler's -d option. The argument to # is a general MonSTB expression and so when using a number this should be in hexadecimal or prefixed by \ for decimal. Thus #10 and #\16 both give the address of line 16 of the program.
- The ASCII load command will change the action of the # operator if the appropriate debug information is available.
- MonSTB has no support for disassembly to disk. Dissassembly to the printer is only available via Alt-P.

# Appendix A

## Reserved Words

---

On the following page is a complete list of the reserved words that are included within HiSoft BASIC.

Normally, these words may not be used as sub-program or label names or as variable names; you can, however, remove a particular reserved word using the ~ compiler option. This is included for maximum compatibility with other versions of the BASIC language. See *Chapter 3* of the *User Manual* for more details. of removing and re-naming HiSoft BASIC reserved words.

Most of these words are built-in HiSoft BASIC language features and, as such, are described in the *Command Reference* chapter; you should note, however, that not all these words will be described directly in this chapter. For example, TO is a reserved word but will not have an entry in the *Command Reference* since it is part of the FOR . . . NEXT syntax.

In addition, there may be some words that are included for compatibility with other languages that are not described in *Command Reference*.

|           |           |           |           |           |
|-----------|-----------|-----------|-----------|-----------|
| ABS       | ACCESS    | ALIAS     | AND       | APPEND    |
| AS        | ASC       | ATN       | AUTO      | BAR       |
| BASE      | BEEP      | BIN\$     | BLOAD     | BREAK     |
| BSAVE     | BYVAL     | CALL      | CALLS     | CASE      |
| CDBL      | CDECL     | CHAIN     | CHDIR     | CHR\$     |
| CINT      | CIRCLE    | CLEAR     | CLEARW    | CLNG      |
| CLOSE     | CLOSEW    | CLS       | COLOR     | COMMAND\$ |
| COMMON    | CONST     | CONTRL    | COS       | CSNG      |
| CSRLIN    | CURDIR\$  | CVD       | CVI       | CVL       |
| CVS       | DATA      | DATE\$    | DECLARE   | DECR      |
| DEF       | DEFDBL    | DEFINT    | DEFLNG    | DEFSNG    |
| DEFSTR    | DIM       | DO        | ELLIPSE   | ELSE      |
| ELSEIF    | END       | ENVIRON   | ENVIRON\$ | EOF       |
| EQV       | ERASE     | ERL       | ERR       | ERROR     |
| EXIT      | EXP       | FEXISTS   | FIELD     | FILES     |
| FILL      | FIX       | FOR       | FORMATD\$ | FORMATIS  |
| FORMATL\$ | FORMATS\$ | FRE       | FREEFILE  | FULLW     |
| FUNCTION  | GB        | GEMSYS    | GET       | GETCOOKIE |
| GOSUB     | GOTO      | GOTOXY    | HEX\$     | IF        |
| IMP       | INCR      | INKEY\$   | INP       | INPUT     |
| INPUT\$   | INSTR     | INT       | INTIN     | INTOUT    |
| IS        | KEY       | KILL      | LBOUND    | LCASE\$   |
| LEFT\$    | LEN       | LET       | LIBRARY   | LINE      |
| LINEF     | LOC       | LOCAL     | LOCATE    | LOF       |
| LOG       | LOG10     | LOG2      | LOOP      | LPOS      |
| LPRINT    | LSET      | LTRIM\$   | MAX       | MID\$     |
| MIN       | MKD\$     | MKDIR     | MKIS      | MKL\$     |
| MKS\$     | MOD       | MOUSE     | NAME      | NEXT      |
| NOT       | OCT\$     | OFF       | ON        | OPEN      |
| OPENW     | OPTION    | OR        | OUT       | OUTPUT    |
| PALETTE   | PCIRCLE   | PCOPY     | PEEK      | PEEKB     |
| PEEKL     | PEEKW     | PELLIPSE  | POINT     | POKE      |
| POKEB     | POKEL     | POKEW     | POS       | PRESERVE  |
| PRESET    | PRINT     | PSET      | PTSIN     | PTSOUT    |
| PUT       | RANDOM    | RANDOMIZE | READ      | REDIM     |
| REM       | REMAINDER | REPEAT    | RESET     | RESTORE   |
| RESUME    | RETURN    | RIGHT\$   | RINSTR    | RMDIR     |
| RND       | RSET      | RTRIM\$   | RUN       | SADD      |
| SCREEN    | SEG       | SELECT    | SGN       | SHARED    |
| SIN       | SOUND     | SPACE\$   | SPC       | SPEEKB    |
| SPEEKL    | SPEEKW    | SPOKEB    | SPOKEL    | SPOKEW    |
| SQR       | STATIC    | STEP      | STICK     | STOP      |
| STR\$     | STRIG     | STRING\$  | SUB       | SWAP      |
| SYSTAB    | SYSTEM    | TAB       | TAN       | THEN      |
| TIME\$    | TIMER     | TO        | TROFF     | TRON      |
| UBOUND    | UCASE\$   | UNTIL     | USING     | VAL       |
| VARPTR    | VARPTRS   | VDISYS    | WAVE      | WEND      |
| WHILE     | WIDTH     | WINDOW    | WRITE     | XOR       |

## Appendix B

# Compiler Options

|    |                                       |          |
|----|---------------------------------------|----------|
| ^  | Output Loadbits                       | output   |
| ~  | Remove reserved words                 | other    |
| !  | Enable variable exports               | standard |
| #  | No "FN"s in libraries                 | standard |
| \$ | Dump Tokens                           | other    |
| %  | Remove inter-sub-program jumps        | advanced |
| &  | Add MonSTB line numbers               | debug    |
| *  | Use old style doubles in files        | advanced |
| .  | Suppress compiler titles              | other    |
| @  | Pre-tokenised file                    | standard |
| [  | Array Warnings                        | standard |
| ]  | Allow undefined sub-programs          | advanced |
| A  | Array Checks                          | standard |
| B  | Break Checks                          | standard |
| C  | Continue on compiler errors           | advanced |
| D  | Use Extended Debug                    | debug    |
| E  | Error Messages                        | standard |
| F  | Output Filename                       | standard |
| G  | Force GEM program                     | advanced |
| H  | Maximum Labels                        | advanced |
| J  | Desk Accessory Size                   | advanced |
| K  | Keep size                             | advanced |
| L  | Leave Size                            | advanced |
| M  | Maths Stack Size                      | advanced |
| N  | Add Line Numbers                      | standard |
| O  | Overflow Checks                       | standard |
| P  | Pause Checks                          | standard |
| R  | Return Stack Size                     | advanced |
| S  | Add Debug information                 | debug    |
| T  | Temporary String Descriptors          | advanced |
| U  | Underlines in variables               | standard |
| V  | Variable Checks                       | standard |
| W  | Suppress Compiler Warnings            | advanced |
| X  | Stack Checks                          | standard |
| Y  | Suppress Default Window               | advanced |
| Z  | Finish without waiting for a keypress | other    |

# Appendix C

## Error Messages

---

### GEMDOS Error Numbers

---

This appendix details the GEMDOS error numbers and their meanings. The error numbers shown are those displayed from within the HiSoft BASIC editor - when calling GEMDOS from your own programs these values are negative.

|    |                       |    |   |
|----|-----------------------|----|---|
| 0  | OK (no error)         | 32 | Invalid function number   |
| 1  | Fundamental error     | 33 | File not found  |
| 2  | Drive not ready       | 34 | Path not found  |
| 3  | Unknown command       | 35 | Too many files open   |
| 4  | CRC error             | 36 | Access denied   |
| 5  | Bad request           | 37 | Invalid handle  |
| 6  | Seek error            | 39 | Insufficient memory   |
| 7  | Unknown media         | 40 | Invalid memory block address                                    |
| 8  | Sector not found      | 46 | Invalid drive   |
| 9  | No paper              | 49 | No more files   |
| 10 | Write fault           | 50 | Disk full (not a GEMDOS error<br>- only produced by the editor) |
| 11 | Read fault            | 64 | Range error   |
| 12 | General error         | 65 | Internal error  |
| 13 | Write protect         | 66 | Invalid program load format                                     |
| 14 | Media change          | 67 | Setblock failure due to growth<br>restrictions                  |
| 15 | Unknown device        |    |   |
| 16 | Bad sectors on format |    |   |
| 17 | Insert other disk     |    |   |



## Runtime Errors

These are errors produced by, or returned from, running compiled programs. They are listed in numeric order for easy reference when messages are not included in the compiled program and for the ERR function. On the next page they are listed in alphabetic order with explanations as to their meaning.

|    |                            |
|----|----------------------------|
| 3  | RETURN without GOSUB       |
| 4  | Out of data                |
| 5  | Illegal function call      |
| 6  | Overflow                   |
| 7  | Out of memory              |
| 9  | Subscript out of range     |
| 10 | Redimensioned array        |
| 11 | Division by zero           |
| 13 | Type mismatch              |
| 16 | String formula too complex |
| 20 | RESUME without error       |
| 23 | Line buffer overflow       |
| 31 | Wrong number of dimensions |
| 50 | FIELD overflow             |
| 51 | Internal error             |
| 52 | Bad channel number         |
| 53 | File not found             |
| 54 | Bad file mode              |
| 57 | Device I/O error           |
| 61 | Disk full                  |
| 62 | Input past end             |
| 63 | Bad record number          |
| 64 | Bad file name              |
| 67 | Too many files             |
| 68 | Device unavailable         |
| 70 | Disk write protected       |
| 75 | Path/file access error     |
| 76 | Path not found             |
| 77 | BREAK pressed              |
| 78 | Fatal bus error            |
| 79 | Fatal address error        |
| 80 | Array not dimensioned      |
| 81 | Sub-program not present    |

## Run-time Errors Alphabetically

### Array not dimensioned

You have used an array that has not been dimensioned. This only occurs when the Array Check Warnings option is enabled. In generally means that you have mis-spelt the array name here or have forgotten to DIMension it..

### Bad channel number

Valid channel numbers are from 1 to 255 inclusive, plus 256 for the default printer channel that is used by LPRINT. In addition windows use channel numbers 257 upwards.

### Bad file mode

You are trying open a channel with an invalid mode string, or trying an input/output operation on a channel that cannot support it, e.g. PRINTing to a file opened for INPUT.

### Bad file name

The filename is invalid, either because it is too long or because it contains invalid characters.

### Bad record number

A record number of 0 is not valid in a PUT or GET statement.

### BREAK pressed

Ctrl-C was pressed during screen input or output. This error cannot be trapped by ON ERROR - if you wish Ctrl-C to be ignored then turn the break checks option off. Pressing Shift-Alt-Help when a program which was compiled to memory is running will also produce this error.

### Device I/O error

A physical error has occurred during an input/output operation.

### Device unavailable

You are trying an operation on a device that cannot support it, such as drawing a circle on a disk file.

### Disk full

## **Disk write protected**

## **Division by zero**

## **Fatal address error**

Address errors are *processor exceptions* caused by a memory access at an odd address, such as PEEKL (&H20011). This error is fatal and cannot be trapped. If a program is compiled to disk this error will produce 3 bombs.

## **Fatal bus error**

Bus errors are *processor exceptions* caused by a memory access at an address in protected or non-existent memory. If you really want to access protected memory use the ST BASIC compatible PEEK or POKE functions. This error is fatal and cannot be trapped, and if it occurs in a program compiled to disk it will produce 2 bombs.

## **FIELD overflow**

A FIELD statement has attempted to use more space than specified in the record size when OPENed, or PRINT# to a random file has filled the current record.

## **File not found**

## **Illegal function call**

This error can be caused by a multitude of things and means a parameter was not in the range required by a function. The module name and line number can be used to track down the error.

## **Input past end**

You have tried to read past the end of a file.

## **Internal error**

This should never happen. It almost certainly means that your program has destroyed part of the heap or the program itself.

## **Line buffer overflow**

A text line in a file is too long for the current line buffer. You can increase the size of this buffer by modifying the long word at SYSTAB+68.

## **Out of data**

A READ has been attempted when no more DATA is available.

## **Out of memory**

Normally this means the heap is full, which is the area of memory used for storing strings and arrays. If the error is in module MALLOC then the GEMDOS pool of memory is full. This can be increased by the use of the Keep and Leave options described in *Appendix B* and *Chapter 3* of the *User Manual*.

## **Overflow**

A result of a numeric calculation is too large to fit into the required type. This can occur when you don't expect it, for example the line

```
test!=32767+1
```

will produce it, as two integers are added together using integer arithmetic. The correct result can be obtained by forcing single-precision arithmetic:

```
test!=32767!+1
```

## **Path not found**

An invalid path was specified in a DOS command.

## **Path/file access error**

You are not permitted to do that operation with that file, such as trying to KILL a read-only file.

## **Redimensioned array**

An existing array must be ERASEd before DIMmed for a second time.

## **RESUME without error**

A RESUME command will only work from within an ON ERROR handler.

## **RETURN without GOSUB**

A RETURN statement was executed but not from a section of the program which had been called with GOSUB. This error will only occur with the stack checks option turned on (see *Appendix B* and *Chapter 3* of the *User Manual*).

## **String formula too complex**

String operations use temporary descriptors for intermediate results, and it is remotely possible to run out and produce this error. You can use the String Descriptors (T) compiler option to increase this.

### **Sub-program not present**

The means that you have called an un-defined sub-program or function. Normally this is checked by the compiler so this error will only occur if you have used the Allow undefined sub-programs option (I) has been used. If you are using the HiSoft GEM Toolkit it means that you've enabled a particular facility (e.g. menus) without providing the appropriate sub-program (ProcessUserMenus in this case).

### **Subscript out of range**

An array subscript used is larger than that specified in the dimensioned statement, negative, or 0 if OPTION BASE 1 is used.

### **Too many files**

A maximum of 255 channels may be opened at once from BASIC. In addition GEMDOS limits the number of simultaneous open disk files to approximately 70.

### **Type mismatch**

A READ was attempted into a numeric variable but the data was found to be a string.

### **Wrong number of subscripts**

An array has been referenced with a different number of dimensions to the number last DIMmed. Unlike other compilers this is a run-time error as dynamic HiSoft BASIC arrays can be REDIMmed with different numbers of dimensions.

## **Compilation Errors**

When the compiler detects an error or something that may be an error (a warning) it generates a message.

The message is prefaced by the error number and followed by the line and file in which the error was detected. In the case of warnings the compiler will continue automatically. After an error it will ask you if you wish to continue. If you type n or N (for no) you will be returned to the editor. If you hit any other key compilation will continue.

When you return to be editor you can use Alt-J to move to the next error with the error message displayed in the status line. If you have a large number of errors the editor may not be able to remember them all. Alt-J goes to the next error regardless of the position of the cursor; it will switch windows is required. To go to a previous error using Ctrl-J. The editor takes account of any insertions or deletions automatically so that unless one error (like a mistake in a definition) has caused multiple errors you should only need to compile once.

There's also the Shift-Alt-J command which finds the next error after the cursor in the current window. It is the appropriate one to use if you have got a number of include files and want to fix all the errors in one file before going on to the next one. You can also use it to find the first error in a file by typing Alt-T (to go to the top) and then Shift-Alt-J.

Normally the error message should be fairly obvious. The following list gives some extra hints for some of the error messages.

Occasionally the compiler will spot errors somewhat later than you might expect. This is usually because the text up to the point it has read is allowed in certain contexts. If you have missed something out at the end of a line, then the error may be detected at the beginning of the next line. Note that, except in the case of missing sub-programs, line numbers or labels, the error in your program can not be *after* the point where the error was detected.

On occasions the compiler will generate more than one error message as a result of a single error in your program; do not be put off by this. If you get confused, just re-compile.

If you have a very badly formed source file, the compiler may slow down considerably. It is probably a good idea to type n to the continue prompt. Incidentally, if you start a compilation of a large program you can break out and returned to the editor using the key combination Alt-Shift-Help when using the integrated compiler.

It is possible that some error messages may have been added to the compiler; only if an error is generated without an error message is this a possible bug in the compiler. If you do get an error number without a message, please tell us.

## **Compiler Error Messages**

In the following list many of the messages give an identifier, reserved word or symbol, this is displayed instead of the % in the message.

### **1 Can't initialise the compiler**

The compiler can't find the HBASIC.LIB file, or there is an attempt to creat too large a label table.

**2 ON... should be followed by GOTO/GOSUB**

May occur if the expression is misformed.

**3 Undefined identifier %**

Occurs within sub-programs and functions if the V option is on. See Appendix B and Chapter 3 of the User Manual.

**4 Unterminated string in data**

**5 Line number/label missing in GOTO/GOSUB % found instead**

This also occurs in ON GOTO/GOSUB/ERROR and RESUME statements if a line number or label is missing.

**6 END DEF assumed**

**7 END SUB assumed**

Error messages 6 and 7 both occur if you attempt to nest sub-program or function definitions.

**8 END DEF where END SUB expected**

**9 END SUB where END DEF expected**

Error messages 8 and 9 are both warnings.

**10 Letter expected after DEFINT etc**

**11 Letter expected after - in DEFINT etc**

**13 Too many ENDS/NEXTs or operand missing**

This error is generated in two different contexts, either when you have too many ENDS or NEXTs or sometimes when omitting one of the operands to a dyadic operator (e.g. \*) in an expression.

**14 Numeric expression expected (not string)**

Many possible causes.

**15 ) expected instead of %**

Many possible causes.

**16 Unterminated string**

**17 Unexpected character in source**

This is a warning, the offending character will be ignored.

**18 FN must be followed by a name.**

You can't use FN on its own as a function name.

**19 Bad line number (% is not positive)**

Or 0 not allowed in this context.

**20 Bad line number (% is too large)**

**21 Bad line number (% has fraction/exponent)**

Or is far too large.

**22 Line number % not found**

**23 Line number % not found**

**24 Parameter (%) may not be used as FOR loop identifier**

Use a local variable instead.

**25 Name (not %) expected in parameter list**

Have you used a reserved word?

**27 CDECL may only be used in DECLARED sub-programs**

But you have tried to use it in a SUB/FUNCTION definition.

**28 ) expected at end of parameter list. % found instead**

Can also be caused by missing out the comma.

**29 Name expected instead of %**

Many possible causes.

**30 Statement only allowed in sub-program or function**

These statements are LOCAL, SHARED, STATIC, EXIT DEF/FUNCTION/SUB.

**32 Label % defined twice**

**33 COMMON must be followed by SHARED**

HiSoft BASIC does not allow any other form of COMMON statement.

**34 Line number/label expected instead of %**

Can occur if an expression is badly formed in an ON...GOTO/GOSUB.

**36 Error during optimisation**

Refer to the message given by the code generator.

**37 Constant % may not be used in parameter definition.**

You have tried to use a constant that you've already defined as a formal parameter in a SUB, DEF FN or FUNCTION definition. Use another name either for the constant or the parameter.

**38 Too many lines in program**

You may not have more than 16383 lines in a single program. See the *User Manual : Concepts* chapter for other compiler limits.

**39 Internal Error (bad source on pass 2)**

Can occasionally be generated by very badly formed source.

**40 Expression mismatch**

Normally caused by missing operands.

**41 END FUNCTION expected before here**

You have attempted to nest a sub-program or function definition inside a FUNCTION.

**42 Expression too complex (too many operators)**

More than twenty operators pending.

**43 Expression mismatch**

**44 Expression too complex**

More than twenty operands pending.

**45 Expression syntax**

**46 Expression syntax**

Normally caused by missing out an entire expression.

**47 FUNCTION or SUB expected after DECLARE**

**48 FUNCTION or SUB declared but not present**

You have used a DECLARE statement for this function but not actually defined with the corresponding SUB or DECLARE statement. If you think you have, check the spellings carefully.

**49 SUB name % used in expression**

**50 Expression mismatch: % is not a unary operator**

**51 Expression mismatch: No unary string minus**

**52 Expression mismatch**

**53 Illegal type combination**

Two strings used with arithmetic operators other than +.



- 54 Illegal type combination (not string)**  
A string expression and a numeric expression have been used with an operator.
- 55 Parameter % appears twice**
- 56 Illegal type combination (not string)**  
String could not be coerced to numeric type.
- 57 % is not a variable or the current function**  
Using other function names is not allowed on the left hand side of assignments.
- 58 Variable is wrong type**  
For example, if you pass a double array instead of an integer array to a sub-program.
- 59 Open bracket expected instead of %**  
Many possible causes.
- 60 Comma expected instead of %**  
Many possible causes. Could be that an expression is mistyped.
- 61 Semi-colon expected instead of %**  
Many possible causes.
- 62 Extra characters at end of statement % is first**  
Perhaps you didn't expect the statement to be over so soon. Check your expressions.
- 63 END DEF assumed at the end of program**
- 64 END SUB assumed at the end of program**
- 65 Code generation failed**  
See the message given by the code generator - the most popular one is disk full!

- 67 Bad option specified**
- 68 ( expected in CALL statement instead of %**  
When using the CALL statement for a sub-program with parameters, you must follow the sub-program name with an open parenthesis.
- 69 Subroutine % not found**  
To CALL a variable use CALL LOC.
- 72 = expected instead of %**  
Many possible causes.
- 74 Unknown meta-command %**  
This error means that you have used a meta-command (after rem \$) that is not supported by HiSoft BASIC. If you are writing a new program check that you have spelt it correctly; if you are porting a program from another compiler, check to find out what it does!
- 75 Structure table full**  
The internal table used by the compiler to store the details of parameter lists, SHARED and local variables is full. If you are importing a lot of global variables many times, make them DIM SHARED.
- 76 LINE must be followed by INPUT not %**  
To plot a line use the LINEF statement.
- 78 FOR variables must be simple variables (not % which is an array)**
- 79 FOR variable % can't be a string**
- 80 TO not % expected in FOR statement**  
Is your initial assignment to the FOR variable correct?



**81 Tokens file too young**

The pre-tokenised .T file that you are using was compiled with an older version of the compiler. Re-make the .T file with this compiler.

**82 Tokens file corrupt**

**83 BASE not % expected after OPTION**

**84 OPTION BASE must be followed by the number 0 or 1 not %**

**85 Tokens file % not found**

**86 Cannot delete % as a reserved word**

As it isn't a reserved word.

**87 Bad extension for output file: .%**

The only valid extensions are .0 (for linkable), .TOS, .TTP, .PRG, .APP, .ACC and .GTP.

**88 % cannot start a statement**

**89 Extra ELSE**

**90 ELSEIF must be followed by THEN not %**

**91 % not allowed after END**

**92 Mismatched NEXT should be %**

**93 END IF expected before here**

**94 END REPEAT expected before here**

**95 NEXT expected before here**

**96 END SELECT expected before here**

**97 WEND expected before here**

**98 LOOP/WEND expected before here**

Errors 93 to 98 may occur if you have omitted the start of the structured statement in the error line.

**99 Name required after SUB or DEF. % found instead**

Is it a reserved word? See Appendix A.

**100 Identifier % redefined**

In sub-program or function definition.

**101 Option value too large**

The only option that may have a parameter bigger than 32767 is the Load Bits (^) option.

**102 Name expected after LET (not %)**

**103 GOTO, THEN or end of line expected after IF expression not %**

**104 Return stack of %K used**

This warning is given when the return stack is more than 200K because in previous versions of HiSoft BASIC the argument to this option was in bytes rather than kilobytes. A return stack of this size is only needed in extremely recursive programs.

**105 Unexpected END SUB or END DEF**

Occurs when not in a function or sub-program definition.

**106 CALL missing at start of statement**

This is a warning. For more information see the *Concepts* chapter in the *User Manual*.

**107 Unterminated control constructs**

Occurs at the end of sub-programs, user-defined functions and sub-programs.

**108 Function or sub-program redefined in library %**

- 109 Unknown REPEAT loop in EXIT**
- 110 Window statement misformed**
- 111 Semi-colon or comma expected**  
Many possible causes.
- 112 REDIM PRESERVE of arrays of more than one dimension not allowed**
- 113 AS or , expected instead of % in OPEN statement**  
Occurs in OPEN, NAME and FIELD statements. Perhaps an expression is wrong.
- 114 CONSTants must be integers not %**
- 115 CONST % can not be assigned to**
- 116 - expected instead of % in graphics GET**
- 117 WINDOW SYSTEM/STOP can only be used in desk accessories**
- 118 % is not a label**  
Your source is probably severely malformed.
- 119 ( expected instead of % in graphics GET or PUT**
- 120 Array expected but ordinary variable % found**
- 121 Argument to VARPTRS must be a subroutine name not %**
- 123 INCLUDE file % not found**
- 125 Value parameter must not be array**

- 127 Library must be string literal not %**  
Have you remembered the quotation marks?
- 128 Cannot open library**
- 129 Library badly formed**  
The HBASIC.LIB file is probably corrupt.
- 130 Too many parameters in statement**
- 131 Sub-program % not found**  
To CALL a variable use CALL LOC.
- 132 GOTO expected after ON ERROR not %**
- 134 SELECT variable must only be a simple variable not %**
- 135 LOOP/WEND expected before here**
- 136 PALETTE USING must be followed by array at %**
- 137 ON/OFF expected after STRIG**

# Appendix D

## The ASCII Table

Here is the 8-bit ASCII representation of the ST/TT's character set:

|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A  | B  | C  | D  | E  | F  |
|------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 0  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 1 1  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 2 2  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 3 3  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 4 4  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 5 5  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 6 6  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 7 7  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 8 8  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 9 9  |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 10 A |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 11 B |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 12 C |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 13 D |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 14 E |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
| 15 F |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |

The ST/TT ASCII set

The most significant four bits of the ASCII representation are shown down the left side whereas the least four significant bits are across the top so that, for example:

4C (4\*16+12=76 decimal) represents L

7B (7\*16+11 = 123 decimal) represents {

# Appendix E

## WERCS HRD file format

.HRD files consist of a header record, any number of variable length data records and then an end-of-file record.

### HRD Header record

| Name       | Size | Meaning   |
|------------|------|---|
| version    | word | 1 at present  |
| autonaming | byte | 1 if auto-naming selected<br>0 if no auto-naming  |
| langflag   | byte | 1 if C,<br>2 if Pascal,<br>4 if Modula-2,<br>8 if FORTRAN,<br>16 if assembler,<br>32 if BASIC |
| autosnap   | byte | 0 if no character-snap<br>1 if half character-snap<br>2 if full character-snap                |
| casing     | byte | 0 if mixed<br>1 if upper<br>2 if lower  |
| autosizing | byte | 1 if auto-sizing<br>0 if no auto-sizing   |
| reserved   | byte | not used at present   |

## HRD Data Record

| Name      | Size   | Meaning  |
|-----------|--------|--|
| type      | byte   | 0 if Form,<br>1 if Menu,<br>2 if Alert,<br>3 if Free String,<br>4 if Free Image,<br>5 if object (rather than tree),<br>6 if end-of-file record,<br>7 if record names a prefix<br>rather than a name. |
| reserved  | byte   | not used at present  |
| treeindex | word   | number of tree   |
| objindex  | word   | if object then object number<br>within tree  |
| name      | varies | Name terminated with a<br>single null  |

# Appendix F

## Linking with C and Assembler

This Appendix is intended for assembly-language and C programmers and details linkage with these languages, together with the memory maps and register usage of compiled programs. If the previous sentence didn't make any sense then stop reading this Appendix now.

### Code Generation

A BASIC source program is converted into true machine-code, there are no P-codes or interpretive run-times. The code produced distinguishes between the program area and the data area. The program area is what is written to the disk or to memory and is position dependent code, relocated either by the GEMDOS loader in the case of disk files or the code generator itself with programs compiled directly to memory. All program code executes in user mode and is compatible with 68010, 68020, 68030 and 68040 processors.

Figure F-1 shows the overall memory map of a compiled program. Note the gap between the program itself, held in the Text segment, and the data area, held in the BSS segment. When a program is run from disk these areas are continuous, but when run from memory using the editor or when in ROM they are not.

### Register Usage

Several registers are committed to special purposes within a compiled program. These are:

#### A3 - Library Pointer

In order to minimise the space and time taken for run-time library calls, register A3 is dedicated to point to the run-time library, with a \$8000 offset to allow a maximum of 64k for the whole library.

Library calls from within the compiled code are of the form

```
JSR    -offset(A3)
```

Library calls from within the library itself do not use A3, they use BSR statements and are resolved by the code generator during compilation.

## A4 - Local Variable Stack Frame

At the very beginning of functions and sub programs a LINK instruction is done to allocate space on the stack for local variables (and function results) and to establish a register that can be used for accessing parameters. Only space for numeric variables are immediately allocated using LINK; arrays and string descriptors are allocated afterwards using a library call. For example a sub-program which has one local integer variable starts with the instruction

```
LINK #-2,A4
```

Functions and sub programs finish with a corresponding

```
UNLK A4
RTS
```

## A5 - Data Area Pointer

The startup code of a compiled program sets up A5 to point to the data area of the program which of course is always RAM. At the start of the area are the run-time globals, followed by the descriptor table (descriptors are described later). Next is the global variable area, used for storing numeric variables. There is a 32k limit on the total size of these globals, but it would take a massive unstructured BASIC program to require such a number of globals. The static arrays follow this; there is no 32K limit on these, but those within this array are accessed more quickly.

## A6 - Maths Stack

This is a special stack used for storing intermediate results of numeric calculations.

## A7 - Machine Stack

The regular machine stack used for return addresses and local variables (using A4).

## D7 - Top of Stack

This register is used to hold the current value from an intermediate expression, be it a single precision numeric value (in the ST version), a long or short integer. For strings D7 is used to hold a pointer to the descriptor for that string.

## D6 - Top of Stack Extension

This register is used in conjunction with D7 to hold long results of type double on the ST version.

## FP0 - Floating point top of stack

In the TT version the floating pointer register FP0 is used to hold intermediate floating point results of both single and double precision.

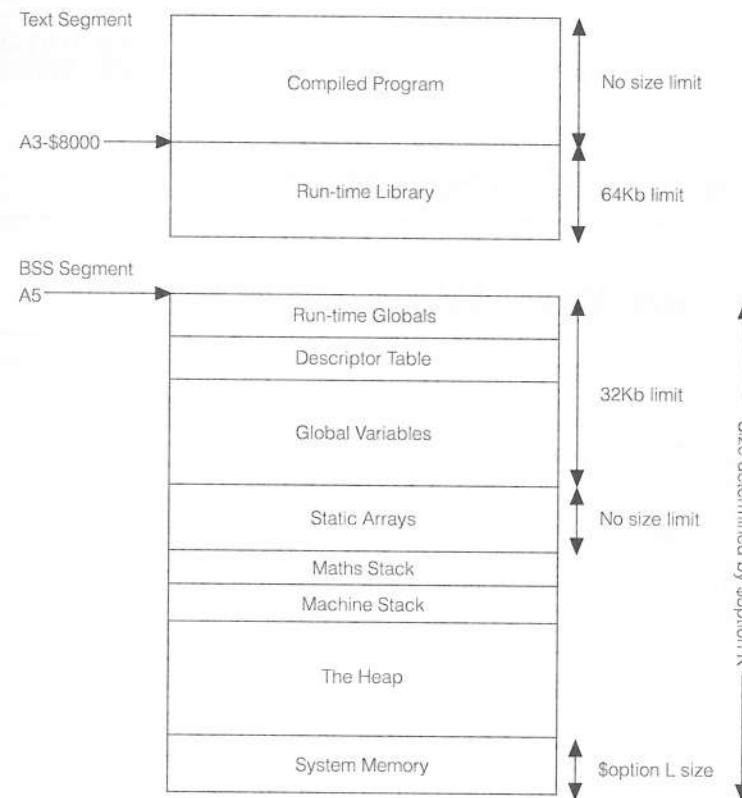


Figure F-1 Compiled Program Memory Map;



## Startup Options

There are three different versions of the startup code, determined by the use of the REM \$option L, REM \$option K and REM \$option J meta commands as described in *Chapter 3* of the *User Manual*.

Note that both non-Desk Accessory versions of the GEM startup code check for the presence of GDOS and, if found, attempt to open a virtual workstation to the screen according to GDOS rules, allowing use of multiple fonts and devices if required. Similarly the WINDOW SYSTEM command will check for the presence of GDOS when it is invoked from a Desk Accessory.

## Low Level Debugging

### Finding Your Way

In addition to using the MonSTB line numbers option, you'll probably also find it easier if you use the Add Line Numbers (N) option which adds instructions of the form

```
MOVE.L #AAAABBBB, (A5)
```

to the program. The hex number AAAA is the physical line number i.e. the line number displayed by the editor, while the hex number BBBB is the last actual line number given in the source. These can be very useful for both finding particular lines and working out where you are in the program.

If the BASIC program has the TRON statement anywhere in it then line numbers are coded using the sequence

```
TRAP #4  
DC.L $AAAABBBB
```

Note that MonSTB also 'understands' about the way that HiSoft BASIC deals with in-line strings. These are really a sub-routine call (to str\_constant) followed by the length of the string and its actual data. Fortunately this is disassembled as:

```
STRING "data"
```

so that you may just skip over it using Ctrl-T.

Note that you may see NOP instructions within the compiled code; don't be alarmed. These are produced by the code generator because on its first pass it leaves room for a JMP absolute instruction when program flow changes, but on pass 2 it notices that the destination is within range for a four byte BRA, so it has to add the NOP (it does not optimise to BRA.S as these take the same time to execute). As the NOPs never get executed there is no speed penalty, but there is a size increase. This is worthwhile; the result is that there are no size limits on compiled programs.

## The Heap and Descriptors

Crucial to the operation of HiSoft BASIC compiled programs is the area of memory known as the *heap*. BASIC is one of two common languages (the other is LISP) that requires dynamic garbage collection. Owing to the way strings in the language work it is necessary to allocate memory for them as required, then, when it runs out, to re-use all the memory no longer needed. This re-allocation is known as *garbage collection*. Many compilers and even some interpreters do not garbage collect and, as a result, certain operations can cause out-of-memory errors even though there is a lot of unused memory left.

HiSoft BASIC has a very advanced memory management system (which is hidden from the user normally) whereby any memory allocation request can cause a garbage collect to occur in order to satisfy the request. The heap itself is a large block of memory from which allocations for string variables and arrays take their memory.

At the bottom of the heap (low memory) are all the string variables, while at the top are all the arrays. Strings work their way upwards, while arrays grow downwards. Should they ever meet, a garbage collect occurs which deletes all unused strings and moves existing ones around as required. It is important to note that arrays never move in memory as a result of a garbage collect; a dynamic array can only move when you REDIM or ERASE any other array; static arrays don't move at all.

As items on the heap are liable, without warning, to move about, ordinary pointers are useless.

For this reason strings and arrays are accessed via descriptors, which exist normally in the global area and which themselves contain actual pointers to the data on the heap. As the memory manager knows where all the descriptors are it can update their pointers when a garbage collect occurs.

Incidentally, the garbage collector itself is very fast; for example it can compact around 350k of fragmented heap in under 2 seconds on a standard 8MHz 68000, so there is never any noticeable delay in the running of a program should it have to garbage collect.

## Memory Formats

### Single-precision Floating Point

These use the Motorola Fast Floating Point (FFP) format. The format is unusual by most standards, but was designed solely with the 68000 architecture in mind, and is as a result very fast. It is the same format as used by ST BASIC, so random access files created with the interpreter should be directly usable within the compiler.



Single Precision

The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa has an implied binary point at bit 32 and thus ranges in value from 0.5 to <1.0. The exponent is held in excess -64. The number zero is represented with 32 bits of zero.

### Double-precision Floating Point

These use the IEEE format double-precision floats, each occupying 8 bytes, i.e. two longs. On the ST version, doubles are actually stored in variables in reverse-long order due to the way the MOVEM instruction works. On the TT version they are stored in their 'natural' order as the FPU is used.



Double Precision Format

The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa has an assumed bit of 1; if present it would be at bit 52. The exponent is held in excess-1023. The number zero is represented with 64 bits of zero.

## Linking or Libraries?

Although this version of BASIC has many powerful features and extensions, occasionally there is a command or function that is not part of standard BASIC. Additionally, a specific task can be very time critical and need all the speed it can get. For these reasons, HiSoft BASIC has the ability to incorporate assembly language subroutines in two ways. You can also link with functions that are written in Lattice C 5.

If you are using assembly language, you have the choice of writing a library or producing linkable code. Libraries have the advantage that once you have written the routine they can be used in exactly the same way as our GEM libraries with no link step required. However code that is used in libraries must be position independent, it cannot access your program's global variables directly and must fit within the 64K of space allocated to the runtime library. If you are modifying your assembly language routine a lot, you'll need to re-make the HBASIC.LIB file after each change. DevpacST version 1.2 or above is required for this.

Using linkable code has the advantages that you can access your global variables directly which can be a big 'win' if the routine is time critical, and the code may be position dependent and as large as you wish. The disadvantage is that you must link your BASIC and assembly code with the CLink linker each time you change either the BASIC or the assembly language.

You'll need DevpacST 3, DevpacTT or Lattice C 5 to produce linkable code.

### Assembler utility routines

Before discussing the mechanics of linkable code and libraries in detail, we will discuss three assembly language routines that can be used when using either method:

#### get\_array

|                 |                 |  |
|-----------------|-----------------|--|
| takes:          | A0              | the array descriptor                                     |
|                 | D0.W            | the number of dimensions the passed array should have    |
| returns:        | A2              | the address of the first element in the array            |
|                 | D4.L            | the total length in bytes taken up by the array elements |
| registers used: | D0,D4,<br>A1,A2 |  |

## get\_string

takes: A0 the string descriptor  
returns: A1 the address of the string  
D4.L the length of the string  
registers used: A0,A1,D4

## make\_string

takes: A0 the string descriptor of a string-variable; the descriptor of a string value will have no effect if passed to make\_string  
A1 the address of your copy of the string  
D4.L the length of the string to be created  
registers used: D0-D4,  
A0-A2

This call allows a string parameter passed by variable to be safely modified by the caller. A value string parameter passed to make\_string will have no effect.

## Linking

As we have seen above, BASIC has a much more complex environment and initialisation procedure than C, so it is only possible for a BASIC program to call C functions; it is *not* possible for C programs to call BASIC code. In addition there are certain conventions which make this process more complex, so calling assembly-language code allows more flexibility.

## External Definition

In order to call C or assembly-language the compiler allows a program to use a special sort of DECLARE statement to indicate that a function or sub-program is written in C or assembler. Using this will automatically cause the compiler to produce linkable (rather than executable) code. If you haven't used the Output to File name (F) compiler option, the name of the output file will have .O added to it (minus the .BGS extension).

### The DECLARE...CDECL statement

The special form of the DECLARE statement is as follows:

```
DECLARE {SUB|FUNCTION} subname CDECL  
[ALIAS "externalname"]  
[(param_list)]
```

The ALIAS clause is optional. The string following indicates the exact external name for the function. If an ALIAS clause is not specified the name (without any type specifier) is made lower-case and an under-score (\_) character inserted at the front. This is suitable for linking with Lattice C compilers. Thus a function called cname% or a sub-program called CNAME will both use the name \_cname, if there is no ALIAS clause.

The parameter list is like that of a BASIC SUB-program or function. That is, it specifies the types of the parameters, how many parameters there are, and whether the parameters are passed by value or reference. Normally the BYVAL keyword should be used to indicate that the parameter is passed by value (or VAL for compatibility with HiSoft BASIC 1 on the Amiga).

It is possible that the name of your external routine will clash with a symbol in the BASIC run-time system. None of the BASIC routines start with an underscore character so we recommend their use on your external names to avoid possible clashes.

The use of such a DECLARE statement will automatically produce linkable code without the need for a compiler option. Compiling a linkable program to memory is permitted to reduce development time but the program will not execute.

## Calling C Functions

C and assembler functions are called in a very similar way to the way standard BASIC sub-programs are called.

If you are using C, call the function or sub-program in the same way as you would in C. Make sure that the types of the parameters in the BASIC DECLARE statement match the sizes of the parameters in the C code (taking particular care with integer sizes). The parameters should be in the same order in both languages.

If a parameter is a pointer and you normally use the C & operator when calling the function, omit the VAL keyword in the parameter list. The address of the BASIC variable will then be passed to the C code.

## Calling Assembly Language Functions

For assembly language, any parameters are coerced to the appropriate type and pushed on the machine stack in reverse order (i.e. C order). Registers A3-A6 must be preserved and any return value should be in D0 (or D0 and D1 if a double, D0=high longword). A string return value should be in D0 and be a pointer to a null-terminated string.

You will need to ensure that your routine 'looks like' a C function. This involves reading parameters from the stack and using an RTS to return. You must leave the parameters on the stack. The code should be assembled to linkable code and the necessary routines exported using XDEF. We recommend that you use the ALIAS facility in the DECLARE statement; otherwise you must use case sensitive code and ensure that the names you export are lower case and start with an underscore.

## Parameters

If numeric parameters are passed by value then their value (16-, 32- or 64-bits) is pushed on the stack. Double-precision parameters are passed in HiSoft BASIC 2 order as required by Lattice C; not in the same order as when using BASIC library.

If variable parameters are passed then the address of the variable is pushed.

Strings and arrays are passed as the address of their descriptor. The descriptor format is private and subject to change so utility routines are provided to convert them into a more useful form.

## Linking

You should end up with a linkable file from the BASIC compiler and a linkable file from the C compiler or assembler. They should be linked together using Clink as supplied with DevpacST3 and Lattice C. You may also need to specify C compiler libraries if your C functions have referred to any.

With Lattice C 5, you should use the compiler options:

`-b0 -r0 -v -w`

which forces 32-bit variable references, 32-bit code and disables stack checking. If when you link there are undefined symbols this normally means your C is calling on a library routine. You will then have to specify the required C library when you link again, but take care that the library code does not assume any register values (e.g. for base-relative globals) or initialisation code (e.g. the Unix file system).

## C-callable Utility Routines

All utility routines may be called from assembly-language but only a few may be called from C, due to register requirements. The C-callable routines can be found in the file hbcutil.o on disk 2 which should be linked with your other code. You must generate stand-alone code to be able to use these routines.

```
long getbasicstr(d,&length)
long d; long length;
```

This should be passed a string descriptor address and the address of where the length should be stored (note that strings may be >64k). The return result is the address of the string which will *not* be null-terminated and you are *not* allowed to write to the string area.

```
long getbasicarray(d,dims,&length)
long d; short dims; long length;
```

This should be passed an array descriptor, the expected number of dimensions and the address of where the length should be stored. The return result is the address of the start of the array contents, guaranteed to be even. The length will be the total length in bytes taken up by the array elements. You are allowed to write to the array area.

## Re-Entrant Code

If you wish a compiled program to be made resident for use under the editor then it is your responsibility for your external routines to follow the necessary rules. We've done our bit...

## Accessing BASIC global variables from assembly

Using the Export Variable names option from the Advanced sub-dialog box in the Compile options box (or via the ! option) the names of your global variables will be exported as absolute symbols so that you can access them via the global register. As the BASIC variable terminators (% ! # \$) are not allowed as parts of normal assembly language names, the terminators are changed to I, L, F, D and S respectively. The names are prefixed with \_ for normal variable and @ for arrays. The variable names are forced to upper case. Conversely sub-program and function names are forced to lower case.

Here are some BASIC names are the corresponding name to use in assembly language:

|         |        |
|---------|--------|
| i%      | _II    |
| fred&   | _FREDL |
| mary!   | _MARYF |
| john%() | @JOHNI |
| MARY!() | @MARYF |

## Linked Examples

Here is an artificially simple example of the process, starting with the BASIC program which uses an external function to compare two short integers:



```
DECLARE FUNCTION cequal% CDECL (VAL p1%,VAL p2%)
PRINT cequal%(1,2),cequal%(10,10)
```

Now the C function:

```
short cequal(a,b)
short a,b
{
return (a==b)
}
```

and the equivalent (Devpac ST 3/DevpacTT ) assembly-language:

```
opt lattice
xdef _cequal
_ cequal move.w 4(sp),d0
cmp.w 6(sp),d0
seq d0
ext.w d0
rts
```

A library is a collection of one or more assembler routines; the exact format of libraries is discussed below. When a library is finished, it is added to HBASIC.LIB using the BUILDLIB program supplied. From there on, your assembly routine can be accessed directly from BASIC by name.

## How to write a Library

A library source does not look much different from that of an ordinary assembly language program. There are several additions which are necessary so that the compiler knows what a routine is called, how many parameters of which type are required and if it is a function or subprogram. In the LIBS folder on your master disks there is a file called LIBDEMO.S. This is an example of the format to use when writing a library. Study this example carefully before writing your own libraries.

DevpacST, version 1.2 and higher, is required to write libraries. You must specify OPT L+,C+ to produce a linkable file with case sensitive labels, or re-install uses these as your default options or on the command line.

A library's source code must conform to a defined format in order for BUILDLIB to be able to incorporate your library into HBASIC.LIB. Failure to conform will not normally be noticed by the assembler but will be reported, sometimes esoterically, by BUILDLIB.

The beginning of every library source has to have the definitions of the routines it contains. The definition commands are defined as macros in the header file LIBRARY.H. This file should be INCLUDED into the source.

```
INCLUDE LIBRARY.H
```

This imports various constants and all the required macro definitions. After this you should define the name of your library; this is the name by which the library will be known to BASIC and need not be the same as its filename. The case of the name is irrelevant e.g.

```
library STUFF
```

Following this, the external references must be specified, e.g.

```
XREF get_string
XREF get_array
XREF make_string
XREF.L gl_scratch
```

These external references are the routines in the BASIC runtime system that your library must call to access arrays and strings and to return strings; note that they are case-sensitive. Constants to be indexed off register A5 are imported using XREF.L.

The names of the routines in your library must follow the external references as external definitions so that BUILDLIB and the compiler can reference them, e.g.

```
XDEF setbit
XDEF getbit
XDEF straint
XDEF baint
XDEF nullterm
```

When referring to these from BASIC, the case of the names is not important. It is however vitally important to remember the sequence in which you specified the names of your routines.

The next lines define the type of routine (subprogram or function), what types of parameters it takes and how many of them, and what type of value is returned (if any).

```
subdef int,vlng setbit
fn_int int,lng getbit
subdef str,aint,lng straint
subdef aint,aint,lng baint
subdef vstr nullterm
```

The reason it is so vital that you remember the sequence of the XDEFs is because the routine definitions have to be in exactly the same order and there must be the same number.

We have created a simple macro-language to cope with the many types of routines and parameters.

| Type   | Meaning   |
|--------|---|
| subdef | This specifies a subprogram, i.e. not a function.   |
| fn_int | This specifies a function that returns a 16-bit signed integer.   |
| fn_lng | This specifies a function that returns a 32-bit signed integer.   |
| fn_sng | This specifies a function that returns a single-precision floating-point number in the Motorola FFP format. |
| fn_dbl | This specifies a function that returns a double-precision floating point number in the IEEE format.         |

There are 15 different types of parameters that can be passed to an assembly language routine. They are:

| Type | Meaning                              |
|------|--------------------------------------|
| int  | a 16-bit signed integer value        |
| lng  | a 32-bit signed long integer value   |
| sng  | a 32-bit FFP single-precision value  |
| dbl  | a 64-bit IEEE double-precision value |
| str  | a string value                       |
| aint | an array of ints                     |
| alng | an array of lngs                     |
| asng | an array of sngs                     |
| adbl | an array of dbls                     |
| astr | an array of strings                  |
| vint | an int-variable                      |
| vlng | a lng-variable                       |
| vsng | a sng-variable                       |
| vdbl | a dbl-variable                       |
| vstr | a string-variable                    |

The first 5 types pass values directly while the last five pass pointers to variables like VAR parameters in Pascal. It is possible to pass a variable as a value parameter; it is *not* possible to pass a value parameter as a variable parameter.

There is also a library command called `option`. This has the same effect as `REM $OPTION` in BASIC and takes the option letters as a parameter.

`option 'UV'`

lets labels containing underscores be valid and forces variable checks. `option` macro calls must occur *after* all sub and fns have been exported, and *before* `libstart`.

When all of the routine definitions are finished, you must tell `BUILDLIB` that your code follows. You do this by specifying:

`libstart`

## The Actual Code

After `libstart`, your code follows. There are two restrictions on the code you write for libraries:

- It must be completely position-independent. If you have Gen version 2 or above, you can specify `OPT P+` to force checks for position independent code.
- Imported runtimes and globals cannot be referenced as part of expressions, so for example

`clr.b gl_scratch+10(a5)`

is not allowed.

A library routine must preserve registers A3-A6. Register A5 must be its normal value when any runtime routines are called.

Parameters are passed to the library routines via the stack in *reverse order*. Value parameters can be found directly on the stack, variable parameters are passed as pointers. Arrays and strings are passed to the library in the form of *descriptors*.

For example, if `vlng`, `aint`, `int` are passed to a routine, they will appear in reverse order on the stack; it would look like this:

| Where                 | Item                                     |
|-----------------------|--|
| <code>(SP).L</code>   | the return address                       |
| <code>4(SP).W</code>  | the INT value                            |
| <code>6(SP).L</code>  | the array descriptor                     |
| <code>10(SP).L</code> | the pointer to the long integer variable |



The int's value can be used directly:

```
MOVE.W 4(SP),D3
```

To get at the long variable is easy as well:

```
MOVE.L 10(SP),A0 the pointer
MOVE.L (A0),D0   the actual value
```

and when you are ready to place the long back from D0 provided you have not altered A0, then

```
MOVE.L D0,(A0)
```

will do just that.

Accessing arrays and strings is a bit more difficult. There are three subroutines in the BASIC runtimes provided just for this purpose: `get_array`, `get_string`, and `make_string` as described above.

### **gl\_scratch**

This is the only area of the runtime globals that a library may use; it is 128 bytes long. `gl_scratch` is referenced off register A5, and whose offset can be imported using

```
XREF.L gl_scratch
```

Register A5 is defined in `LIBRARY.H` by a register equate as `global`. If your routine needs more memory, you will have to use stack space. *Under no circumstance* should you use any part of the heap!

Don't forget that strings in HiSoft BASIC can be larger than 64k, thus length parameters are always long in size.

To get the aint passed in the example, you must do the following:

```
MOVE.L 6(SP),A0
MOVEQ.L #1,D0 check if a single dimensional array
BSR get_array
```

A2 now contains the address of the first element of the array, D4.L contains the total length of the array.

If an array has more than one dimension, the dimensions are arranged in a linear fashion. For example, `Array(0,0)` is the first, then comes `Array(0,1)` until `Array(0,end)`, then comes `Array(1,0)` etc.

## **Strings and Things**

The next example, `nullterm`, shows the use of `get_string`, `make_string`, and `gl_scratch`. Since libraries do not support a function that returns a string, a different way is used to return strings. The example below takes a string variable, copies it into `gl_scratch`, null-terminates it and returns the string to the variable.

```
nullterm
        MOVE.L 4(SP),A0      get the descriptor
        BSR get_string
        CMP.L #128,D4        check for size
        BLT.S OKSiz
        MOVEQ #127,D4        reduce size to fit
        LEA gl_scratch(global),A0
        MOVEA.L A0,A2        save address of copy

        MOVE.L D4,D5          save string length
        SUBQ.W #1,D4          subtract 1 for DBF
Loop     MOVE.B (A1)+,(A0)+   copy string
        DBF D4,Loop

        MOVEA.L A2,A1          get the address back
        MOVE.L D5,D4           get the length back
        CLR.B 0(A1,D4.L)      null terminate

string
        ADDQ.W #1,D4          a byte was added
        MOVE.L 4(SP),A0       the target

descriptor
        BRA make_string      return the string

and
*
                                then back to BASIC
```

## **Returning values from Functions**

When a function returns a value to BASIC, it does so by putting the value in the register named `tos` (which is actually register D7). This applies to `fn_ints`, `fn_lngs`, and `fn_sngs`. This looks like:

```
MOVE.W D0,tos returns a short integer
MOVE.L D0,tos returns 32 bits
```

When returning a 64-bit double, the register `tos2` (actually D6) is used in addition to `tos`. The high 32-bits (exponent, sign and some of the mantissa) go into `tos`, while the low 32-bits of the mantissa go into `tos2`.

## Library Format Summary

To summarise, the basic format of library source code is:

|           |                    |                        |
|-----------|--------------------|------------------------|
| opt       | L+,C+              | linkable, case         |
| dependent |                    |                        |
| include   | LIBRARY.H          |                        |
| library   | library_name       | defines name           |
| XREF      | runtime1, runtime2 | runtimes needed        |
| XREF.L    | global1, global2   | globals needed         |
| XDEF      | mysub1             |                        |
| XDEF      | mysub2             | a list of exports      |
| XDEF      | myfn3              |                        |
| subdef    | mysub1             |                        |
| subdef    | mysub2             |                        |
| fn_int    | myfn3              | matched list for BASIC |
| option    | 'vun'              | (optional)             |
| libstart  |                    |                        |

*your code here*

You source file should be assembled to a .BIN file, then linked into the BASIC library by running BUILDLIB.

## Using BUILDLIB

BUILDLIB is used to add user libraries to HBASIC.LIB, the compiler's library. The user library file must have the .BIN extension, as it is in linkable format.

BUILDLIB is a .TTP file, i.e. it takes a command line. On this command line should be listed the names of the files to be added to HBASIC.LIB, without their .BIN extensions specified. If no filenames are given, HBASIC.LIB will consist of only the standard runtimes.

BUILDLIB expects to find the HBASLIB.BIN file in the current directory; this file contains the BASIC runtimes. It creates the file HBASIC.LIB in the current directory. If you are using the desktop you will need to copy BUILDLIB .TTP to the same directory as the the libraries.

There are a few options that can be used with BUILDLIB. These should be at the start of the command line, prefaced by a - sign.

- T Produce HBASICTT.LIB rather than HBASIC.TTP for use with the stand-alone compiler.
- Z Don't wait for a key on termination

- Q Produce the standard library with gemdos, bios, xbios, gemvdi, gemaes and menu. This command line and may not be followed or preceded by additional file names.
- L List progress. This displays what BUILDLIB is doing as it does it, enabling you find which library is causing an error.

It can be easier to find symbol errors (e.g. undefined symbol) by using LinkST (supplied with DevpacST and Lattice C) to link HBASIC.BIN and your library. Any executable file produced by the linker should be deleted as it is not executable.

For example, double-clicking on BUILDLIB.TTP then entering

-Q STESOUND

will produce a full version of library with the STESOUND library in addition.

-L DEMO TEST

will cause the libraries contained in DEMO.BIN and TEST.BIN to be added to the runtimes from HBASLIB.BIN in HBASIC.LIB; the -L option dumps the narrative on the screen.

-ZT GEMDOS BIOS XBIOS

will produce a library for the TT with just the low-level parts of the operating system.

Note that if you run BUILDLIB from within the HiSoft BASIC editor using Run Other or the Tools menu, the new library created will not be used by programs you compile until you leave the editor and re-enter it. This is because the editor reads HBASIC.LIB into memory when it starts up and won't notice if it changes it in the meantime.

## STESOUND library

This library uses the STE sound hardware which is also available on the TT. Before using this library you should check that the appropriate hardware is present using:

```
IF (GETCOOKIE("_SND") AND 2) = 0 THEN
PRINT "Sorry this program requires DMA SOUND hardware"
STOP -1
```

### SUB DMAcontrol(BYVAL newvalue%)

This sets the STE sound hardware's control register. The appropriate values are:

|   |                     |
|---|---------------------|
| 0 | Disable DMA sound   |
| 1 | Do sound once       |
| 3 | Do sound repeatedly |

### SUB DMAsound(BYVAL startadr&, BYVAL endadr&, BYVAL mode%)

This sets the start and end addresses of the sample address for the DMA sound hardware together with the sound mode. The `mode%` gives the sample rate (see the table below) and also whether mono or stereo is to be used. Add 128 in the values in the table for mono.

|   |         |
|---|---------|
| 0 | 6258Hz  |
| 1 | 12517Hz |
| 2 | 25033Hz |
| 3 | 50066Hz |

### SUB MWout(BYVAL mask%, BYVAL data%)

Writes the values `mask%` and `data%` to the STE MICROWIRE interface. The full generality of this is not used, but it can be used to set the volume/tone and mixing of the STE DMA sound at present. This command issues the command immediately but waits for the command to be complete before returning.

### SUB MasterVolume(BYVAL db)

This sets the DMA sound hardware's master volume to be db decibels. db should be between -80 and 0; the hardware supports this in 2 db steps.

### SUB LeftVolume(BYVAL db)

This sets the DMA sound hardware's left volume to be db decibels. db should be between -40 and 0; the hardware supports this in 2 db steps.

### SUB RightVolume(BYVAL db)

Sets the DMA sound hardware's right volume to be db decibels. db should be between -40 and 0; the hardware supports this in 2 db steps.

### SUB Treble(BYVAL db)

This sets the DMA sound hardware's treble control to be db decibels. db should be between -12 and +12; the hardware supports this in 2 db steps and 0 is flat.

### SUB Bass(BYVAL db)

This sets the DMA sound hardware's bass control to be db decibels. db should be between -12 and +12; the hardware supports this in 2 db steps and 0 is flat.

### SUB SetMix(BYVAL mixvalue)

This controls the DMA Sound hardware's mixing with the standard ST hardware sound. The following values of `mixvalue` are supported:

|   |           |
|---|-----------|
| 0 | -12 db    |
| 1 | Mix       |
| 2 | Don't mix |

# Appendix G

## Desk Accessories

---

HiSoft BASIC can compile programs into a special form allowing the creating of desk accessories, usually reserved for languages such as C or Pascal. However HiSoft BASIC 2 will let you write desk accessories with very few explicit AES or VDI calls or even none at all when using the HiSoft GEM Toolbox (HGT). However when you use these short-cuts that HiSoft BASIC provides your program won't necessarily be able to cope with resolution changes or windows that are left open when another program terminates. Please don't expect them to be!

When testing your desk accessories don't be surprised if you crash the machine, because of a bug in your program. If you are testing on an auto-booting hard disk make sure that you know the procedure for disabling this and starting from floppy.

### Get Ready

---

Unlike previous versions of HiSoft BASIC there's no need to use an additional HBASIC.LIB file when compiling desk accessories.

Any program wishing to be compiled to desk accessory from should have the line

```
REM $OPTION Jxx
```

where xx is equivalent to the `Keep` size in Kilobytes. It is used to specify the whole workspace size of the desk accessory while it runs. This will change the extension of the compiled program to .ACC.

The minimum value allowed is 15k, while 20k is recommended for smaller programs. This may also be set using the DA Size option from the Advanced box within the Compile dialog box. If you use this method ensure that you are using a different HISOFTED.INF file from the one that use for normal programs, perhaps by using a different directory for them.

Attempting to compile to memory will produce the message program buffer full.

## Commands to use

Here are some routines that you will find useful when programming desk accessories.

```
FUNCTION menu_register%  
    (BYVAL id%,BYVAL name$);
```

This GEMAES library function should be called early in the initialisation of the program and registers the accessory on the Desk menu. *id%* is the applications id, which can be found using the expression

```
PEEKW(PEEK(GB+4)+4)
```

and *name\$* is the name that appears in the Desk menu, normally starting with two spaces and no longer than 20 characters. The result of the function is the menu item identifier, from 0 to 5, or -1 if there is no room on the menu for the name.

The startup code for an accessory is quite different to that of a normal compiled .PRG file in that little initialisation is done, the only AES call made is `appl_init`.

Note that `da_before` and `da_after` that were used in HiSoft BASIC 1 are no longer present or required.

## WINDOW SYSTEM

This command, which is built in to the standard BASIC library, opens the BASIC's window system virtual workstation and internal tables so that you can then use `PRINT` etc. This should be called after your program receives its open message.

## WINDOW STOP

This closes the virtual workstation that was opened using `WINDOW SYSTEM`. You should always call this before your program returns to its event loop if you've used `WINDOW SYSTEM`.

## Structure of a Desk Accessory

An accessory should have the structure of any event-driven GEM program, based on a main loop of `evnt_multi` or `evnt_mesag`. When the accessory is chosen from the Desk menu a message of type `AC_OPEN` (40, not 30 as often documented) will be received. The message `AC_CLOSE` (41, not 31) will be sent either when a GEM application terminates or the screen resolution is changed from the Desktop.

If you are using BASIC windows, then you use `WINDOW SYSRTEM` when you receive your `AC_OPEN` message

A compiled desk accessory is rather different to an ordinary program. For example an accessory can never *terminate* - if it does the machine will crash. Should a program stop for any reason, such as with a run-time error, the message will be printed at the bottom of the screen and a keypress awaited in the normal way, then the accessory will become effectively dead until the machine is re-booted.

## Restrictions

Desk Accessories under TOS suffer from a major technical disadvantage; although they give every appearance of being a separate process, they actually can't allocate their own resources once the system has booted. Every time a desk accessory calls GEMDOS to obtain some memory, whether directly or indirectly it, the memory it receives is marked as owned by the current application - as such when that application terminates all that memory is de-allocated *before* the desk accessory is sent its `AC_CLOSE` message. This explains why desk accessories behave strangely is you leave their windows open when you quit an application..

There's also the further difficulty that when you change resolutions the memory that the program loaded at boot time (including any resource file that is loaded then) is removed without telling the desk accessory giving bombs when you change resolutions.

If you want to write a professional program that will survive resolution changes etc, you'll have to forgo many features that you've taken for granted in HiSoft BASIC and you'll need to call the operating system directly. If you can live with the problems that are described above then you'll find that you can actually use almost everything that you are used to.

The following commands should *not* be used within a desk accessory:

CHAIN, COMMAND\$, RUN

## Appendix H

# Program Profiling

---

### Introduction

---

Profiling is a way to find out where your program is spending its time. This is particularly useful when a program is finished and you want to improve its performance.

There are two distinct steps involved in obtaining a program profile: firstly the program itself must be modified to create a file containing the required information, and run; next the profiling analysis program is run. It presents the profiling information in an easy to understand form.

### Creating the Profile

---

Profiling a program is quite easy. All you must do is insert the line

```
rem $pron
```

where you would like profiling to start. If you want to stop profiling, at any particular line you must state:

```
rem $proff
```

Turning the profile off is not a necessity; on the other hand, a profile is of more value when it measures the time critical areas of a program as accurately as possible. When your program is compiled extra information is appended to the executable file which is used by the analysis program. When your program has finished running, a file containing the profile information is created on the current drive in the current directory with the extension .PRF.

You must compile to disk when using profiling.

The format of the .PRF file is straightforward: it is ASCII, the first line is the name of the executable being profiled and the remainder of the lines are of the form:

```
line_number,times_sampled
```



The first number is the line number in the source, the second is the number of times this line was found to be executing. If you use the `$include` meta-command, `line_number` will be the line number processed by the compiler, not necessarily the line number within an included file. Note that a comma separates the two numbers so that `INPUT#` can be used to access the data.

The extra information in the executable file tells the analysis program about any use of `$include` and where sub-programs and functions start and finish.

## Analysing the Profile

Supplied with the compiler is a utility called `PROFILE.TTP`. This is the analysis program for profiling data. Run `PROFILE.TTP` by double-clicking on it from the desktop or by selecting it from the Tools menu in the editor. The parameter passed to the program is the name of the `.PRF` file (with or without extension) you wish to analyse. The profiler will automatically read the executable file - this must be in the place where it was created.

The profiler will then prompt you

How many lines?

Enter a number *n*; the analyser will show the *n* most frequently executed lines together with the percentage of time spent and the number of times the line was sampled, depending on the number you enter; the default (just hit [Return]) is the top 12 lines.

If you have used sub-programs and/or functions, Profile will prompt you again:

How many SUBs and FNs?

Again enter a number *n*; this will show the top *n* subprograms and functions; the default is 8.

Finally you will be asked:

Output to:

The default is the screen. You can also have the analysis data put into a text file; all you have to do is specify the name and extension of the file the information is to be written to.

If the output is going to the screen, you can use `Ctrl-S` and `Ctrl-Q` to pause and continue output.

# Appendix I

## TT Specifics

HiSoft BASIC takes full advantage of the Atari TT computer with support for the 68030 & 68881/2 processors; and also many library functions and sub-programs that allow you to access the new operating system supplied with the TT and MegaSTE computers.

## TT version of the compiler

The ST version of the compiler runs on the TT and the editor can take advantage of the extended screen modes. It produces code that will run on any machine in the ST/TT family.

The TT version of the compiler works in a similar way to the standard version except that it uses and generates 68030 instructions for faster and more compact code and generates inline code for the 68882 maths co-processor. The compiler itself and the code produced requires at least a 68020 and 68881 combination.

The code for the 68882 uses the full extended precision on chip and stores intermediate values on the maths stack as doubles regardless of whether they are stored in memory as singles or doubles. In general the TT version is more accurate than the software routines. Rather than storing singles as Motorola Fast Floating Point (FFP), IEEE singles are used as supported by the chip directly. These have different accuracy and precision to FFP numbers.

As such, you should not expect the same answers for numeric calculations on both compilers; if you are developing for both the ST and the TT you need to check the accuracy of your calculations on both machines.

We could have made the TT version produce results that are much closer to those produced by the ST version but the code produced would have run much slower as in line code could not have been produced. One floating point intensive benchmark runs 100 times faster with the TT version than it does on the ST version on the same machine.

For commercial developers who want their products to run on both machines, rather than providing an auto-sensing version of the system that would use software or hardware floating point according to the system we suggest that you provide two versions as we have done with HiSoft BASIC as this really lets the TT version take advantage of the new hardware.

## TT Libraries

We have included the additional operating system calls that were introduced with TT TOS as part of the standard libraries. These cover access to the new video facilities, memory allocation etc.

As these routines are included in the standard libraries you can use them with the ST compiler; thus if you want one version of your program to run on both the ST and TT, you can still use the TT TOS facilities when running on a TT.

The STESOUND library also works on the TT.

## TT Debugger

Unlike the other parts of the package MonSTB does not work properly on the TT because it assumes that it has a 68000 processor; the screen handling routines that write directly to screen RAM do not 'know' about the TT resolutions and so MonSTB can not be used with them.

Thus you must use MonTTB; in addition to the features described above you can also access the additional TT MMU and floating point registers and the disassembler will disassemble the floating point and additional 68030 instructions. Register FP0 is used by the compiler to store the current floating point intermediate result.

## Index

^ 343

.BH file details, resource editor  
306

### ■ A

address  
of subprogram 197  
of variable 196  
screen 220

address error 335

alert boxes  
in GEM toolbox 274

APPEND 119

appl\_find 248

appl\_read 248

appl\_write 248

APPLBLK 302, 303

Array not dimensioned 347

ASCII Table 363

AUX 119

### ■ B

Bad channel number 347

Bad file mode 347

Bad file name 347

Bad record number 347

bconin& 218

bconmap& 226

bconout% 218

bconstat% 218

bcostat% 219

binary 7

BIOS

input 80  
output 122

bioskeys 224

BITBLK structure 302

blitmode% 226

blitter mode 226

BLOCKMOVE 220

BREAK pressed 347

Breakpoint, debugger (see  
Debugger, breakpoint)

BUILDLIB 384

bus error 335

### ■ C

CASE statement 162

cauxin% 210

cauxis% 212

cauxos% 212

cauxout% 211

cconin& 210

cconis% 211

cconos% 211

cconout 210

cconrs 211

cconws 211

CHANGE\_HANDLE 232

CHK instruction 335

CIRCLE 125

clear

screen 21

close% 214

cnecin% 211

COLOR 123

command line 24

Command Reference 1

Compilation Errors 350

CON 119

copy memory, debugger 329

cpynos% 212

- cprnout% 211
- crawcin% 211
- crawio% 211
- create
  - directory 109
- create% 214
- current directory 30
- cursconf% 223
- cursor
  - set position 98
- cursor keys
  - debugger 316
- cursor position 133
- CVHANDLE 232
- CVI 31
- CVL 31
- CVS 31
- D**
- DATA. 145
- dcreate% 213
- ddelete% 213
- Debugger
  - abort 321
  - address, set 314
  - Auto Load Source 328
  - Automatic Prefix Labels 328
  - baseconvert 315
  - breakpoint 318
    - conditional 319
    - count 319
    - GEMDOS 321
    - kill 320
    - permanent 319
    - remove 320
    - set 314, 319, 320
    - show 320
    - simple 318
    - stop 319
  - command summary 331
  - copy memory 329
  - cursor keys 316
  - disassembly window 313
  - edit 314
  - executing programs 323
  - expressions 309
  - find 325
  - Follow Traps 327
  - font 314
  - GEMDOS breakpoint 321
  - hints 332
  - history 321
  - Ignore Case 327
  - interrupt program 318
  - labels 311
  - line numbers 328
  - load binary 322
  - load program 322
  - load source 323
  - lock window 315
  - low resolution 312, 317, 323
  - memory layout 336
  - memory window 313
  - numbers 310
  - preferences 327
  - print window 315
  - register
    - set 315
    - window 312
  - registers 311
  - Relative Offsets 327
  - running program 324
    - go 324
    - instruction 324
    - slowly 324
    - until 325
  - running programs 323
  - save binary 322
  - screen mode 317
  - screen switching 317, 327
  - search memory 325
  - Show Line Numbers in Source 328
  - single step 323
  - skip 324
  - split window 316
  - symbols 311
  - Symbols Option 328
  - terminate 321
  - Top Of RAM 329
  - user screen 317
  - windows 311
    - commands 314
    - disassembly 313
    - edit 314
    - lock 315
    - memory 313
    - print 315
    - register 312
    - source code 313
    - split 316
    - type 316
    - zoom 316
  - zoom window 316
- Debugging
  - lowlevel 370

- DEF SEG 127, 131
- Default, flag type 298
- delete
  - directory 155
  - file 86
- desk accessories 389
  - in GEM toolbox 283
- desktop
  - in GEM toolbox 280
- Desktops, resource editor 303
- Device I/O error 347
- Device unavailable 347
- DevpacST 378
- dfree 213
- dgetdrv% 212
- dgetpath 216
- dialog boxes
  - in GEM toolbox 274
- directory
  - changing current 15
  - create 109
  - delete 155
- Disk full 347
- Disk write protected 348
- Division by zero 348
- DMAcontrol 229
- dmaread& 226
- DMAsound 230
- dmawrite& 227
- dosound& 225
- drawing mode 237
- drvmap% 219
- dsetdrv% 211
- E**
- Editable, flag type 298
- EgetPalette% 228
- EgetShift% 227
- ELLIPSE 129
- ERR 52
- EsetBank% 227
- EsetColor% 228
- EsetGray% 229
- EsetPalette 228
- EsetShift%% 227
- EsetSmeat% 229
- evnt\_button 249
- evnt\_dclick 250
- evnt\_keybd 248
- evnt\_mesag 249
- evnt\_mouse 249
- evnt\_multi 250
- evnt\_timer 249
- Exceptions, re-install, debugger 330
- executing programs, debugger 323
- Exit, flag type 298
- expressions, debugger 309
- F**
- Fatal address error 348
- Fatal bus error 348
- fattrib% 215
- fdatetime 217
- fdelete% 214
- fdup% 215
- fforce 215
- fgetdta& 213
- FIELD overflow 348
- file
  - delete 86
  - rename 114
- file handling
  - create 119
  - length of file 99
  - open 119
- File not found 348
- file numbers 65
- file selector
  - in GEM toolbox 281
- files
  - attribute bits 215

- find, debugger 325
- Flag States (See Resource Editor, states)
- Flags, resource editor 298
- flopfmt% 222
- floprate% 226
- floprd% 221
- flopvr% 223
- flopwr% 221
- Font
  - debugger 314
- fopen% 214
- form\_alert 254
- form\_center 255
- form\_dial 254
- form\_do 254
- form\_error 255
- formatted output 138
- fread& 214
- frename% 217
- fseek& 215
- fsel\_exinput 258
- fsel\_input 258
- fsetdta 212
- fsfirst% 217
- fsnext% 217
- FUNCTION menu\_register% 390
- functions 38
- fwrite& 214

## ■ G

- G\_BOX 296
- G\_BOXCHAR 297
- G\_BOXTEXT 296
- G\_BUTTON 297
- G\_FBOXTEXT 297
- G\_FTEXT 297
- G\_IBOX 297
- G\_ICON 297

- G\_IMAGE 297
- G\_PROGDEF 297
- G\_STRING 297
- G\_TEXT 296
- G\_TITLE 297
- GDOS 232
- GEM toolbox 267
  - desk accessories 283
  - desktop routines 280
  - dialog and alerts 274
  - file selector 281
  - image handling 282
  - image windows 278
  - InitMenuSystem 273
  - InitResourceFile 268
  - mainloop 268
  - menus 273
  - object handling 269
  - object windows 278
  - StartProgram 268
  - StopProgram 268
  - text windows 278
  - windows 276

## GEMAES

- Events 248
- File Selector 258
- Forms 254
- Graphics 255
- Menus 251
- Message Passing 248
- Objects 252
- Resource Files 263
- Scrap Directory 257
- Shell Routines 264
- Utility routine 248
- Windows 258

## GEMAES Library 247

## GEMDOS Error Numbers 345

## GEMDOS Library 210

## GEMVDI

- Attributes 237
- Control 232
- Drawing Primitives 234
- Enquires 244
- Mouse & Keyboard State Functions 243
- Raster Functions 242
- get\_array 373
- get\_string 374

- getbpb& 219
- getmpb 217
- getrez% 220
- gettime& 224
- giaccess% 224
- gl\_scratch 382
- GOSUB 153
- graf\_dragbox 255
- graf\_growbox 256
- graf\_handle 256
- graf\_mkstate 257
- graf\_mouse 257
- graf\_movebox 255
- graf\_rubberbox 255
- graf\_shrinkbox 256
- graf\_slidebox 256
- graf\_watchbox 256
- graphics
  - box 5
  - circle 18
  - ellipse 46
  - fill 59
  - filled area 235
  - filled circle 125
  - filled ellipse 129
  - get area 67
  - line 93
  - lines 234
  - markers 234
  - pie slice 235
  - read pixel colour 130
  - rounded box 236
  - rounded filled box 236
  - text 234

## ■ H

- HBASIC.LIB 384
- HBASLIB.BIN 384
- heap 64
- hexadecimal 75
- HGT, see GEM toolbox 267
- HGTLoop, GEM toolbox routine 268
- Hide, flag type 298

- hints
  - debugger 332
  - resources 303
- history
  - debugger 321
- HRD file
  - format 365

## ■ I

- ICONBLK structure 301
- ikbdws 224
- Illegal function call 348
- illegal instruction 335
- images
  - in GEM toolbox 278, 282
- INDIRECT, flag type 298
- InitMenuSystem, GEM toolbox routine 273
- initmous 220
- InitResourceFile, GEM toolbox routine 268
- Input past end 348
- INPUT# 120
- INPUT\$ 120
- Internal error 348
- internal shell 265
- interrupt program, debugger 318
- INTERSECTION 248
- iorec& 222

## ■ J

- jdisint 224
- jenabint 224
- joystick 174, 178, 179
- justify
  - left 103
  - right 157

## ■ K

- KBD 119
- kbdvbase& 225

kbrate% 225  
kbshift% 219  
keytbl& 222

## ■ L

labels, debugger 311  
LASTOB, flag type 298  
leading spaces 104  
libraries  
    format summary 384  
LIBRARY statement 209  
Line buffer overflow 348  
Linking or Libraries? 373  
Linking with C and d Assembler 367  
load binary, debugger 322  
load program, debugger 322  
load source, debugger 323  
LOF 120  
LOG10 100  
logbase& 220  
low resolution  
    debugger (see Debugger, low resolution)  
lower case 88  
LSET 110  
LST 119

## ■ M

machine code  
    calling 11  
    libraries 92  
maddalt& 212  
make\_string 374  
malloc& 216  
match  
    string within string 84  
maths co-processor 70  
maximum 105  
mediach% 219  
memory

    free 64  
memory layout, debugger 336  
Memory Map 369  
Menu& 265  
menu\_bar 251  
menu\_ichack 251  
menu\_ienable 251  
menu\_text 252  
menu\_tnormal 251  
menus  
    in GEM toolbox 273  
mfpint 222  
mfree% 216  
MID 119  
midi 222  
midiws 222  
minimum 108  
MKD\$ 31, 110  
MKI\$ 31  
MKL\$ 31, 110  
MKS\$ 31, 110  
mode  
    of screen 161  
mshrink% 216  
MWout 230  
mxalloc& 215

## ■ N

numbers, debugger 310  
numeric to string conversion 63  
NVMAccess% 227

## ■ O

ob\_flags 294  
ob\_head 294, 296  
ob\_height 294  
ob\_next 294, 296  
ob\_spec 294  
ob\_state 294

ob\_tail 294, 296  
ob\_type 294  
ob\_width 294  
ob\_x 294  
ob\_y 294  
objc\_add 252  
objc\_change 253  
objc\_delete 252  
objc\_draw 252  
objc\_edit 253  
objc\_find 252  
objc\_offset 253  
objc\_order 253  
object, struct defintion 294  
objects  
    Flags (See Resource Editor:flags)  
    in GEM toolbox 269

Octal 115

offgibit 224

ongibit 224

operators  
    debugger 309

Out of data 348

Out of memory 349

OUTPUT 119

Overflow 349

## ■ P

PARMBLK structure 302

Path not found 349

Path/file access error 349

PEEKB, PEEKL, PEEKW 128

pexec% 216

physbase& 220

POS function 133

preferences  
    debugger (see Debugger, preferences)

PRESERVE 146

PRINT 206

print formatting 63  
PRINT# 120, 137  
PRINT# USING 137  
processor 69  
proff 393  
Profiling 393  
programming with resources 294  
pron 393  
protobt 223  
PRT 119  
pterm 216  
pterm0 210  
ptermres 213  
puntaes 226  
PUT  
    file I/O statement 142  
    graphics statement 143

## ■ R

Radio Button, flag type 298

Rainbow TOS 262

RANDOM 119

random numbers 156

random& 222

RANDOMIZE 156

read

    keyboard without echo 79

Redimensioned array 349

registers, debugger 311

Rename file 114

REPEAT...END REPEAT 148

Resource Editor

    APPLBLK 302

    BASIC 306

    BITBLK 302

    Border

        thickness 299

    Box 285, 296

    BoxChar 285, 297

    BoxText 285, 296

    Button 285, 297

    C 306

    Checked 291



- child 296
- colour
  - names 300
  - numbers 300
  - word 299
- Crossed 291
- Default 289
- Disabled 291
- Editable 290
- example program 306
- Exit 289
- FBoxText 286, 297
- Flag Types 289
- Flags
  - summary 292
- FText 288, 297
- grand child 296
- greyed. 291
- Hide 290
- hints 303
- HRD file
  - format 365
- IBox 288, 297
- Icon 288, 297
- ICONBLK Structure 301
- Image 288, 297
- mistakes 304
- New Desktops 303
- ob\_head 296
- ob\_next 296
- ob\_tail 296
- objects
  - states 299
  - struct definition 294
  - summary 292
- Outlined 291
- parent 296
- PARMBLK 302
- ProgDef 288, 297, 303
- Radio Button 290
- root 296
- Selectable 289
- Selected 291
- Shadowed 291
- sibling 296
- String 288, 297
- TEDINFO 286, 290
- TEDINFO Structure 300
- Template. 286
- Text 288, 296
- Title 288, 297
- Touch Exit 290
- UnHide Children 291
- UserDef 288
- Valid 286
- WTEST 306

- Resources, programming with 294
- RESUME without error 349
- return code 176
- RETURN without GOSUB 349
- rsconf 222
- RSET 110
- rsrc\_free 263
- rsrc\_gaddr 263
- rsrc\_load 263
- rsrc\_obfix 264
- rsrc\_saddr 264
- rtblk 225
- Running program
  - debugger 324
- Running programs, debugger 323
- Runtime Errors 346
- rwabs% 218

## ■ S

- save binary, debugger 322
- scrdmp% 223
- screen
  - address 220
- screen mode 220
- screen switching, debugger 317
- screndump 126
- scrip\_read 257
- scrip\_write 258
- search
  - within string 84
- search, debugger 325
- Selectable, flag type 298
- SELECT...END SELECT 162
- setcolor% 221
- setexc% 218
- setpalette 221
- setpath% 213
- setprt% 225
- setscreen 221

- settime 213, 224
- shel\_envrn 265
- shel\_find 264
- shel\_get 264
- shel\_put 265
- shel\_read 264
- shel\_write 265
- shell buffer 264
- shift keys 111
- SIN function 166
- single step, debugger 323
- skip, debugger 324
- SOUND 198
- source code, debugger 313
- SPEEKB, SPEEKL, SPEEKW 170
- square root 172
- ssbrk 220
- StartProgram,GEM toolbox routine 268
- StopProgram,GEM toolbox routine 268
- STRIG 174
- string
  - address of 160
- String formula too complex 349
- strip spaces 104, 158
- Sub-program not present 350
- subprogram
  - address of 197
- subprograms
  - calling 10
- Subscript out of range 350
- super% 212
- supexec 226
- sversion% 213
- symbols
  - debugger 311

## ■ T

- TEDINFO structure 300
- terminate, debugger 321
- tgetdate% 212
- tgettext 213
- tickcal% 219
- Too many files 350
- Touch Exit, flag type 298
- trailing spaces 158
- TRAPV instruction 335
- tsetdate 213
- Type mismatch 350

## ■ V

- v\_arc 235
- v\_bar 235
- v\_circle 235
- v\_clrwk 233
- v\_clsvwk 233
- v\_clswk 233
- v\_contourfill 235
- v\_ellarc 236
- v\_ellipse 236
- v\_ellpie 236
- v\_fillarea 235
- v\_get\_pixel 243
- v\_gtext 234
- v\_hide\_c 243
- v\_justified 236
- v\_opnvwk 233
- v\_opnwk 232
- v\_pieslice 235
- v\_pline 234
- v\_pmarker 234
- v\_rbox 236
- v\_rfbbox 236
- v\_show\_c 243
- v\_updwk 233



variable  
  address of 196  
vq\_color 244  
vq\_extnd 244  
vq\_key\_s 244  
vq\_mouse 244  
vqf\_attributes 245  
vql\_attributes 245  
vqm\_attributes 245  
vqt\_attributes 245  
vqt\_extent 246  
vqt\_fontinfo 246  
vqt\_name 246  
vqt\_width 246  
vr\_recfl 235  
vro\_cpyfm 242  
vrt\_cpyfm 243  
vs\_clip 234  
vs\_color 237  
vsc\_form 244  
vsf\_color 241  
vsf\_interior 241  
vsf\_perimeter 241  
vsf\_style 241  
vsf\_updat 242  
vsl\_color 238  
vsl\_ends 238  
vsl\_type 238  
vsl\_udsty 238  
vsl\_width 238  
vsm\_color 239  
vsm\_height 239  
vsm\_type 239  
vst\_alignment 240  
vst\_color 240  
vst\_effects 240  
vst\_font 240  
vst\_height 239

vst\_load\_fonts 234  
vst\_point 239  
vst\_rotation 240  
vst\_unload\_fonts 234  
vswr\_mode 237  
vsync 225

## ■ W

wind\_calc 262  
wind\_close 259  
wind\_create 258  
wind\_delete 259  
wind\_find 261  
wind\_get 260  
wind\_new 262  
wind\_open 259  
wind\_set 261  
wind\_update 261  
windows  
  debugger (see Debugger, windows)  
  in GEM toolbox 276, 278  
WRITE# 120  
Wrong number of subscripts 350  
WTEST 306

## ■ X

xbtimer 225

## ■ Z

zero divide 335  
zoom window, debugger 316